IRIDIS ALPHA THEORY

ROB HOGAN

Contents

Making Planets for Nigel

> 17 February 1986
>
> Redid the graphics completely, came up with some really nice looking metallic planet structures that I'll probably stick with. Started to write the `GenPlan` routine that'll generate random planets at will. Good to have a C64 that can generate planets in its spare time. Wrote pulsation routines for the colours; looks well good with some of the planet structures. The metallic look seems to be 'in' at the moment so this first planet will go down well. There will be five planet surface types in all, I reckon, probably do one with grass and sea a bit like 'Sheep in Space', cos I did like that one. It'll be nice to have completely different planet surfaces in top and bottom of the screen. The neat thing is that all the surfaces have the same basic structures, all I do is fit different graphics around each one.
>
> — Jeff Minter's Development Diary in Zzap Magazine[?]

Making planets is easy.

When making a planet, ensure you perform each of the following simple steps in the order given below.

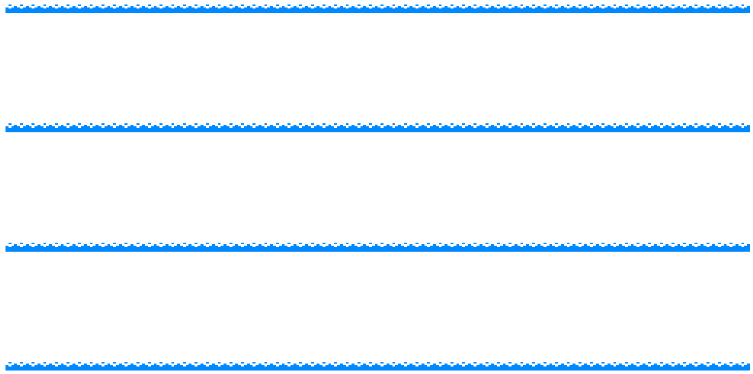


Figure 1.1: Step One: Add the sea across the entire surface of the planet.

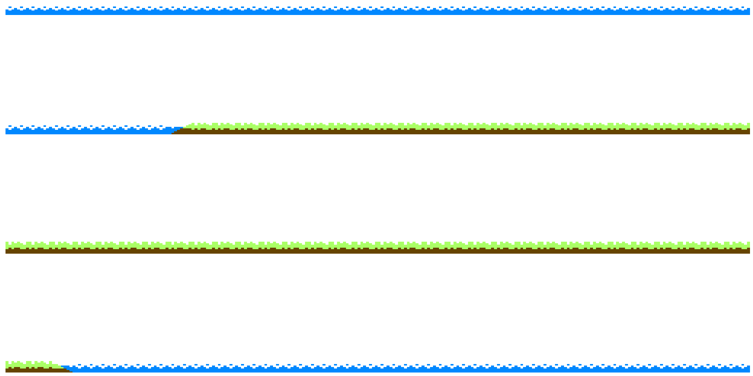


Figure 1.2: Step Two: Insert a land mass at least 32 bytes and at most 128 bytes long.

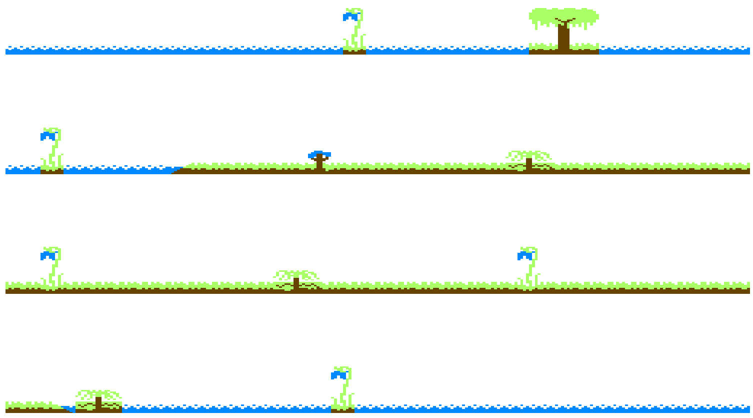


Figure 1.3: Step Three: Add a random structure every 13 to 29 bytes.

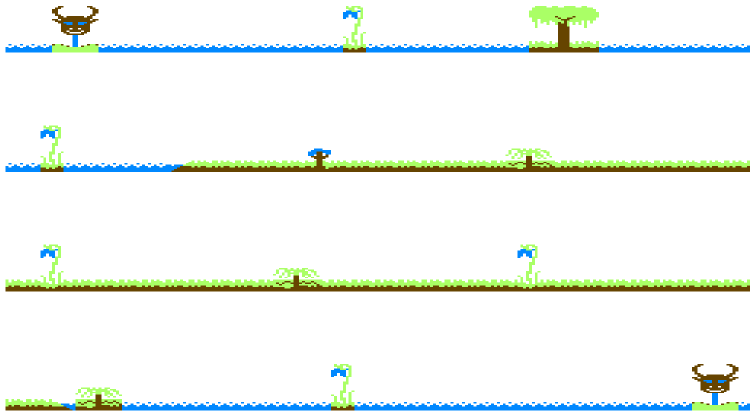


Figure 1.4: Step Four: Add warp gates at the beginning and end of the planet surface.

Now you have not just a layout for one planet, but a layout for all five.

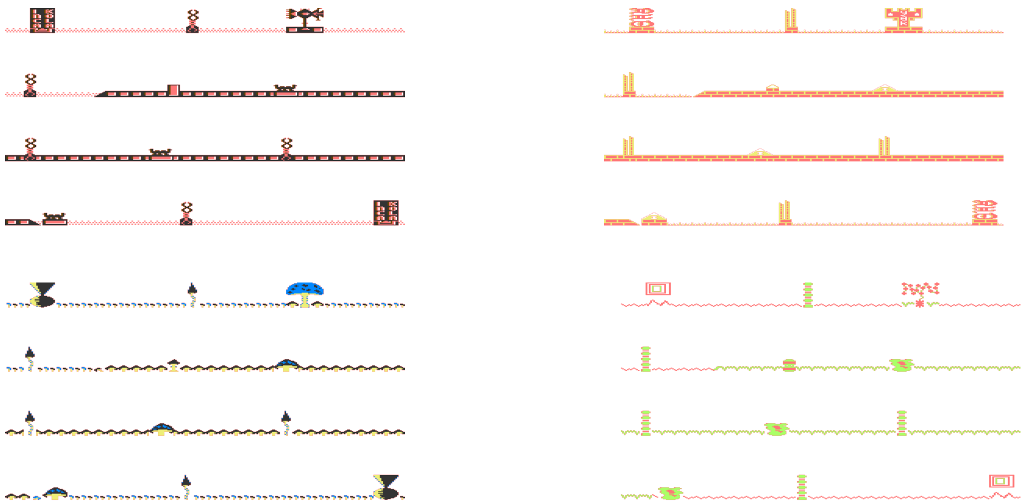
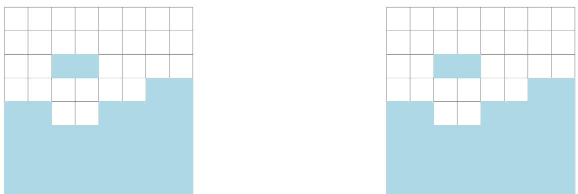


Figure 1.5: A layout that will suit all the planets in your life.

But making planets isn't all simple steps and big picture decisions. There are also trifling details for the little people to wrestle with.

1.0.1 Step One: Creating the Sea

Making a sea is very easy. You come up with a character than can be repeated 1024 times to fill the surface of the planet.



((1)) planet1Charset $40 **((2))** planet1Charset $42

Figure 1.6: There are two characters used for creating the sea and they're both the same! This will make more sense when we look at the land, where they are different.

Figure 1.7: planet1Charset Sea

The bit that needs explaining is how you define the character. If it was a simple bitmap then we could imagine the character as 8 rows of 8 bits and where a bit is set to 1 you color that pixel in. That is not the case. You can see how the bits are actually set below:

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0
1	0	0	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0

Figure 1.8: planet1Charset $40 representing a tile of sea.

Look closely at the picture above and you should see how it works. What is happening is that we fill two adjacent cells with blue when together they form the value `10`. So we create graphic characters not with a simple bit-map but with a map of bit pairs. Each pair of bits is treated as a unit giving us four units on each row. Maybe it's intuitively obvious that `00` means 'blank' or 'background' but I've pointed that out to you now just in case.

```
planet1Charset
        .BYTE $00,$00,$20,$02,$8A,$AA,$AA,$AA    ;.BYTE $00,$00,$20,$02,$8A,$AA,$AA,$AA
                                                 ; CHARACTER $40
                                                 ; 00000000
                                                 ; 00000000
                                                 ; 00100000      *
                                                 ; 00000010          *
                                                 ; 10001010    *   * *
                                                 ; 10101010    * * * *
                                                 ; 10101010    * * * *
                                                 ; 10101010    * * * *
```

Listing 1.1: Character `$40` representing the sea as it is defined in the source code. A full eight bytes are required to define each character so not cheap.

Is that all there is to it? No. Before we look at how me might color things other than blue, let's look at how we color them with the big blue brush we have so far. The first thing we do is clear down the entire surface of the planet:

```
; of the screen. The neat thing is that all the surfaces have
    the same
; basic structures, all I do is fit different graphics around
    each one."
;
    ------------------------------------------------------------------

GeneratePlanetSurface
        LDA #<planetSurfaceData
        STA planetSurfaceDataPtrLo
        LDA #>planetSurfaceData
        STA planetSurfaceDataPtrHi

        ; Clear down the planet surface data from $8000 to
    $8FFF.
        ; There are 4 layers:
        ; Top Layer:    $8000 to $83FF - 256 bytes
        ; Second Layer: $8400 to $87FF - 256 bytes
        ; Third Layer:  $8800 to $8BFF - 256 bytes
        ; Bottom Layer: $8C00 to $8FFF - 256 bytes
        LDY #$00
ClearPlanetHiPtrs
        ; $60 is an empty character and gets written to the
    entire
        ; range from $8000 to $8FFF.
        LDA #$60
ClearPlanetLoPtrs
        STA (planetSurfaceDataPtrLo),Y
```

Listing 1.2: The surface data is stored from $8000 to $8FFF. This code overwrites it all with the value $60 which is an empty bitmap.

```
        .BYTE $00,$00,$00,$00,$00,$00,$00,$00    ;.BYTE $00,$00,$00,$00,$00,$00,$00,$00
                                                 ; CHARACTER $60
                                                 ; 00000000
                                                 ; 00000000
                                                 ; 00000000
                                                 ; 00000000
                                                 ; 00000000
                                                 ; 00000000
                                                 ; 00000000
                                                 ; 00000000
```

Listing 1.3: The empty character bit map (all zeroes) used to overwrite the surface before populating it.

With the planet surface cleared out (overwritten with all $60s) we can now.. overwrite it all again with sequences of $40,$42. No, that's not right. We're only overwriting the bottom layer - the surface layer - this time. This is the layer that contains the land and/or sea and it lives between $8C00 and $8FFF which if your hexadecimal arithmetic is better than mine you will realize is 1024 bytes ($400 in hex).

```
        LDA planetSurfaceDataPtrHi
        CMP (#>planetSurfaceData) + $10
        BNE ClearPlanetHiPtrs

        ; Fill $8C00 to $8FFF with a $40,$42 pattern. These are the
        ; character values that represent 'sea' on the planet.
        LDA #$8C
        STA planetSurfaceDataPtrHi
WriteSeaLoop
        LDA #$40
        STA (planetSurfaceDataPtrLo),Y
        LDA #$42
        INY
        STA (planetSurfaceDataPtrLo),Y
        DEY
        ; Move the pointers forward by 2 bytes
        LDA planetSurfaceDataPtrLo
        CLC
        ADC #$02
        STA planetSurfaceDataPtrLo
        LDA planetSurfaceDataPtrHi
        ADC #$00
```

Listing 1.4: Filling the entire bottom surface of the planet with $40,$42 which gives us the sea. Our next step is to overwrite some of this with land.

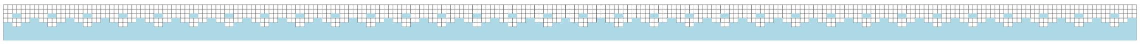


Figure 1.9: That sea again. Our work so far.

1.0.2 Step Two: Creating the Land

Is that all there is to it? Painting things with blue? No.

There are other possible values aside from 10 and 00 that we could use to paint colors. We could also have 11 and 01. This is useful since we want to color things in with more than one color. We have blue assigned to 10 on Planet 1, while for the land we can use two other colors: 11 which we will assign 'green' and 01 which we will assign 'brown'. We can assign whatever colors we like but we can only choose three, not counting the background. This is the kind of limitation you run into when you only allow two bits for assigning possible colors.

((1)) planet1Charset $41 **((2))** planet1Charset $43

Figure 1.10: Planet 1 Land uses two different characters that alternate to generate the land surface.

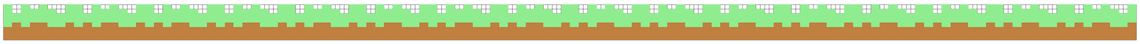


Figure 1.11: planet1Charset Land

The location and length of the landmass is randomly generated with a couple of constraints: it must be at least 128 bytes and not more than 256 bytes from the start of surface and it must be at least 32 bytes and not more than 150 bytes long. The result is that the planet surface will be mostly sea since the entire surface is 1024 bytes long.

Picking a random number between 128 and 256 is slightly convoluted in assembly:

```
; Pick a random point between $8C00 and $8FFF for
; the start of the land section.

; Get a random number between 0 and 256 and store
; in A.
JSR PutRandomByteInAccumulatorRegister
; Ensure the random number is between 128 and 256.
AND #$7F ; e.g. $92 becomes $12.
```

Listing 1.5: Convoluted

Neat Little Trick?

```
;
   ------------------------------------------------------------------
; PutRandomByteInAccumulatorRegister
;
   ------------------------------------------------------------------
PutRandomByteInAccumulatorRegister
```

Listing 1.6: Neat

This little snippet's job is to return a quasi-random byte for use in the planet generation routines. To achieve this, it does something quite fiendish that is more or less unhead of in modern programming: it mutates itself.

When called for the first time it loads a value from the address at `randomPlanetData` to the accumulator. On first run `randomPlanetData` points to the address `$9ABB` which contains the value `$42`:

```
randomPlanetData
        .BYTE $42,$E4,$3F,$94,$4E,$29,$B0,$59
        .BYTE $2C,$FE,$7F,$B2,$40,$9B,$63,$2B
```

Listing 1.7: Not Quite Random Bytes

Before returning this value as its result it alters itself by changing `randomPlanetData` to reference `$9ABC` (`INC randomIntToIncrement`). In other words, it increments the pointer. In the assembly listing we make `randomIntToIncrement` reference the position that holds `randomPlanetData` by positioning it one byte before and adding a 1 to shift is reference beyond the byte holding `LDA` to `randomPlanetData`.

Every time the routine is called it increments the reference again so that the next time it will pick up whatever lies in the bytes beyond `9ABB`. The results it returns are never truly random, but random enough to permit the procedural generation of planets that they're used for.

— A

With a random start position selected, a similar convolution is performed to choose the length of the land mass:

```
        ; in the planet between $8C00 and $8FFF and store it in
        ; planetPtrLo/planetPtrHi
        JSR StoreRandomPositionInPlanetInPlanetPtr

        ; Randomly generate the length of the land section, but
        ; make it at least 32 bytes and not more than 150.
        JSR PutRandomByteInAccumulatorRegister
```

Listing 1.8: A convolution

Since the random number we get can be anything between `$00 - $FF` (i.e. 0 and 255) and we want a number that's between 0 and 128 we need to do a bitwise `AND` to mask out Bit 7 which by itself is 128.

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$FC	1	1	1	1	1	1	0	0
$7F	0	1	1	1	1	1	1	1
Result; $7C	0	1	1	1	1	1	0	0

AND'ing $FC and $7F gives $7C (124).

With the position and length selected we can start laying turf. We don't just plop down our basic land tiles. Posh and proper means giving the shore of the land its own look and feel. This we have in the characters `$5C` and `$5E` in our character set:

$5C $5E $5D $5F

Figure 1.12: Character tiles for the left shore ($5C,$5E) and the right shore ($5D,$5F).

Now we can put the rest of the land down:

```
        LDA #$5E
```

Listing 1.9: Write pairs of `$41,$43` for the main land mass.

And finally the right shore:

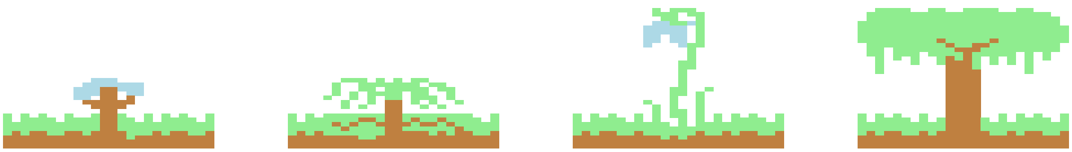
```
        STA (planetPtrLo),Y
        DEC planetSurfaceDataPtrLo
        BNE DrawLandMassLoop

        ; Draw the right short of the land, represented by the
    chars in
        ; $5D/$5F.
        INY
        LDA #$5D
```

Listing 1.10: Drawing the right shore.

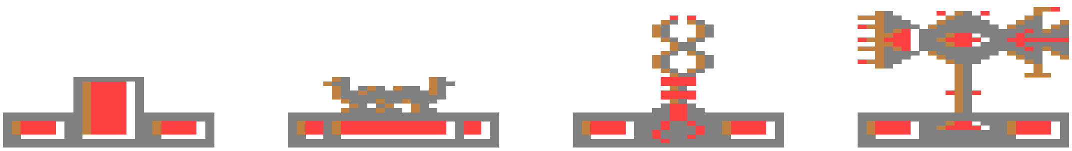
1.0.3 Step Three: Structures Structures Structures

The routines for adding structures to the planet are the opportunity to observe some assembly language cleverness. For each structure we draw we have to decide two things: where to drop it on the surface and what type of structure to draw. Apart from the Warp Gates, there are four structure types available.



((1)) planet1Charset littleStructureData | ((2)) planet1Charset mediumStructureData | ((3)) planet1Charset nextLargestStructure | ((4)) planet1Charset largestStructureData

Figure 1.13: The four possible structure types for Planet 1.



((1)) planet2Charset littleStructureData | ((2)) planet2Charset mediumStructureData | ((3)) planet2Charset nextLargestStructure | ((4)) planet2Charset largestStructureData

Figure 1.14: The four possible structure types for Planet 2.

You may be getting the sense that there is a sort of economy at work here. The structures are effectively the same for each planet, but with the textures swapped out. Your intuition is correct, the structures are only defined once and the same definition does regardless of which planet we're painting:

```
        JSR GeneratePlanetStructures

        RTS

mediumStructureData  .BYTE $65,$67,$69,$6B,$FF
                     .BYTE $64,$66,$68,$6A,$FE
largestStructureData .BYTE $41,$43,$51,$53,$41,$43,$FF
                     .BYTE $60,$60,$50,$52,$60,$60,$FF
                     .BYTE $49,$4B,$4D,$4F,$6D,$6F,$FF
                     .BYTE $48,$4A,$4C,$4E,$6C,$6E,$FE
```

Listing 1.11: The definitions of three of the structures above each of which serves all five planets.

The $FF at the end of each line serves as a sentinel for the drawing routine to know that the subsequent bytes are for the next layer 'up'. The $FE is a terminator, indicating there is no more data for the structure.

Drawing a structure is relatively straightforward so we'll cover that briefly first. Drawing the littlest structure provides the most compact example of the technique:

```
;
   ------------------------------------------------------------------
; DrawLittleStructure ($7486)
;
   ------------------------------------------------------------------
DrawLittleStructure
        ; Start iterating at 0.
        LDX #$00
DrawLSLoop
        ; Get the byte in littleStructureData pointed to
        ; by X.
        LDA littleStructureData,X
        ; If we reached the 'end of layer' sentinel, move
        ; our pointer planetPtrHi to the next layer. The
        ; BNE 'stays on the same layer' by jumping to
        ; LS_StayonSameLayer if the current byte
        ; is not $FF.
        CMP #$FF
```

```
        BNE LS_StayonSameLayer
        ; Switch to the next layer.
        JSR SwitchToNextLayerInPlanet
        ; SwitchToNextLayerInPlanet incremented X for us
        ; so continue looping.
        JMP DrawLSLoop

LS_StayonSameLayer
        CMP #$FE
        ; If we read in an $FE, we're done drawing.
        BEQ ReturnFromDrawingStructure
        STA (planetPtrLo),Y
        ; Increment Y to the next position to write to.
        INY
        ; Increment X to get the next byte to read in.
        INX
        ; Continue looping.
        JMP DrawLSLoop
```

Listing 1.12: The littlest structure has only two layers.

Given that we're only writing 4 bytes this is a lot of code. As we will see there are separate routines for each of the structures and unfortunately for our search for evidence of coding genius they're all identical. So this is a pretty open-and-shut case of code duplication. It would have been more compact to rationalize them down to a single function and use a pointer to the structure data instead of repeating almost verbatim the same assembly code for each structure.

```
        INX
ReturnFromDrawingStructure
        RTS

;-------------------------------------------------------
; DrawMediumStructure ($74B1)
;-------------------------------------------------------
DrawMediumStructure
        LDX #$00

DrawMSLoop
        LDA mediumStructureData,X
        CMP #$FF
        BNE b74C0
        JSR SwitchToNextLayerInPlanet
        JMP DrawMSLoop

b74C0   CMP #$FE
```

```
        BEQ ReturnFromDrawingStructure ; Return
        STA (planetPtrLo),Y
        INY
        INX
        JMP DrawMSLoop

;-------------------------------------------------------
; DrawLargestStructure ($74CB)
;-------------------------------------------------------
DrawLargestStructure
        LDX #$00

DrawLargeStructureLoop
        LDA largestStructureData,X
        CMP #$FF
        BNE b74DA
        JSR SwitchToNextLayerInPlanet
        JMP DrawLargeStructureLoop

b74DA   CMP #$FE
        BEQ ReturnFromDrawingStructure ; Return
```

Listing 1.13: `DrawMediumStructure` and `DrawLargestStructure` are identical to each other and to `DrawLittleStructure` and `DrawNextLargestStructure`.

The cleverness comes a little earlier so let's console ourselves with that. When we've chosen a position to draw our structure we need to pick a type of structure at random. The secret to this is to store the addresses to our regrettably repetitive draw routines in a pair of arrays.

```
        ; The routine contains an 'RTS' so does the returning
        ; for us.

;Jump table
```

Listing 1.14: A 'jump table' containing the addresses to our draw routines. The address for `DrawLittleStructure` happens to be $7486 so we store $74 in the first byte of `structureSubRoutineArrayHiPtr` and $86 in the first byte of `structureSubRoutineArrayLoPtr`.

With this in place our routine consists of getting a random number between 0 and 3, then using that as in index to pick out a value at the same position from `structureSubRoutineArrayLoPtr` and `structureSubRoutineArrayHiPtr`. We then store those values in `structureRoutineLoPtr` and `structureRoutineHiPtr` respectively. We now have a pointer to one of our draw routines at `structureRoutineLoPtr` which we can jump to with the simple command: `JMP (structureRoutineLoPtr)`.

```
;---------------------------------------------------------------
; DrawRandomlyChosenStructure
;---------------------------------------------------------------
DrawRandomlyChosenStructure
        ; Pick a random positio to draw the structure
        JSR StoreRandomPositionInPlanetInPlanetPtr

        ; Run the randomly chose subroutine, one of:
        ; DrawLittleStructure, DrawMediumStructure,
        ; DrawLargestStructure, DrawNextLargestStructure
        ; to draw a structure on the planet surface

        ; Pick a random number between 0 and 3
        JSR PutRandomByteInAccumulatorRegister
        ; AND'ing with $03 ensures the number is between
        ; 0 and 3.
        AND #$03
        ; Move the number to the X register.
        TAX
        ; Use the random number to pick and draw a structure.
        LDA structureSubRoutineArrayHiPtr,X
        STA structureRoutineHiPtr
        LDA structureSubRoutineArrayLoPtr,X
        STA structureRoutineLoPtr
        ; With the address of the routine we've chosen copied
        ; to structureRoutineLoPtr, we jump to that address and
```

Listing 1.15: `DrawRandomlyChosenStructure` picks a random position and a random draw routine to use at that position.

Rinse and repeat this for the length of the map and we get a surface with sea and land that is dotted with structures of different types.

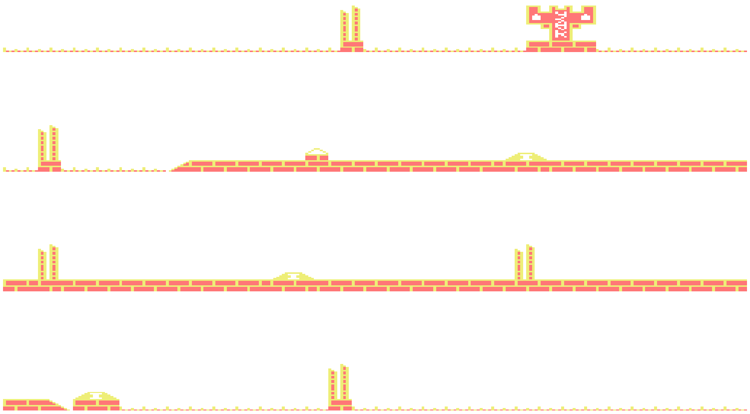
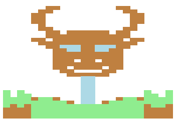


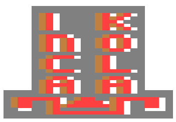
Figure 1.15: Planet 3 once `DrawRandomlyChosenStructure` has finished its business.

1.0.4 Step Four: Add the warp gate

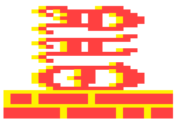
Our final step is to add the warp gate.



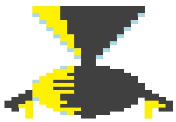
((1)) planet1Charset warpGateData



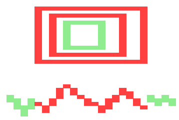
((2)) planet2Charset warpGateData



((3)) planet3Charset warpGateData



((4)) planet4Charset warpGateData



((5)) planet5Charset warpGateData

Figure 1.16: The warp gates on each planet.

There's something funny here I haven't figured out yet. The routine for drawing the warp gate draws it twice. Yet each level has only one warp gate. Each one gets an initial position of $F1 and $05 respectively. This is used by `StoreRandomPositionInPlanetInPlanetPtr` to point to a position on the surface where the warp gate is drawn.

```
        JMP GenerateStructuresLoop

        ; Draw the two warp gates
DrawWarpGates
```

```
        LDA charSetDataPtrLo
        BEQ GenerateStructuresLoop

        ; Draw a warp gate at the end of the map.
        LDA #$F1
        STA charSetDataPtrHi

        JSR StoreRandomPositionInPlanetInPlanetPtr
        JSR DrawWarpGate
        DEC charSetDataPtrLo

        ; Draw a warp gate at the start of the map.
        LDA #$05
        STA charSetDataPtrHi
```

Listing 1.16: Why does it draw 2 warp gates when there's only 1? Haven't figured this out yet..

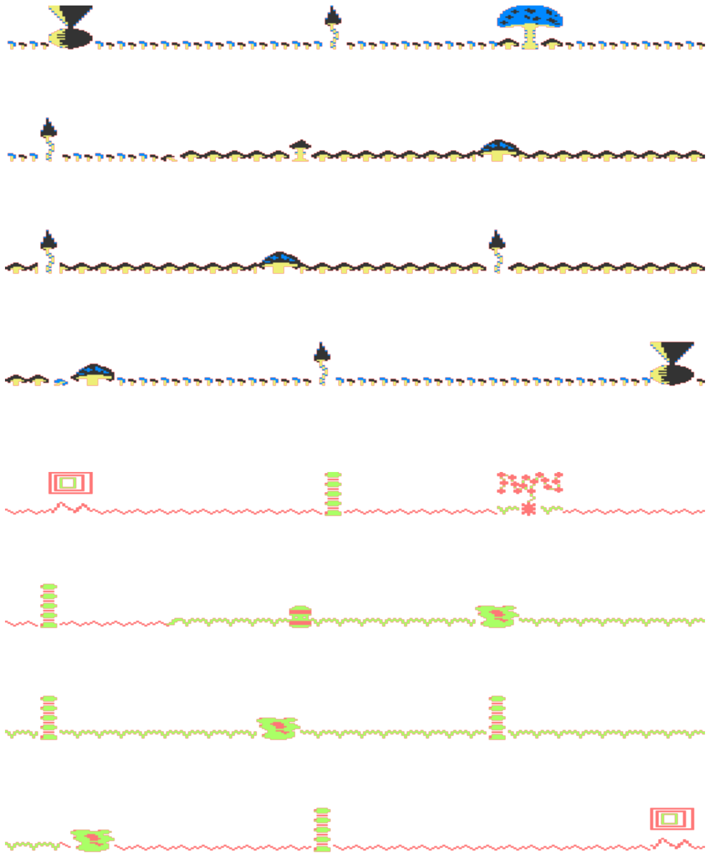


Figure 1.17: The final surfaces for Planets 4 and 5.

1.0.5 Inactive Lower Planet

When the lower planet is inactive a surface with land, sea, and a warp gate is displayed. This doesn't reuse any of the logic described above. Instead it is generated from some customized data in the routine `DrawLowerPlanetWhileInactive`.

```
;-------------------------------------------------------------
; DrawLowerPlanetWhileInactive
; Draws the lower planet for the early levels when it isn't
; active yet.
;-------------------------------------------------------------
DrawLowerPlanetWhileInactive
        LDA lowerPlanetActivated
```

```
        BEQ b6047

        LDX #$28
DrawLowerTextLoop
        LDA textForInactiveLowerPlanet - $01,X
        AND #$3F
        STA SCREEN_RAM + LINE18_COL39,X
        LDA #WHITE
        STA COLOR_RAM + LINE18_COL39,X
        DEX
        BNE DrawLowerTextLoop

        LDX #$28
DrawInactiveSurfaceLoop
        LDA surfaceDataInactiveLowerPlanet,X
        CLC
        ADC #$40
        STA SCREEN_RAM + LINE14_COL39,X
        DEX
        BNE DrawInactiveSurfaceLoop

        LDX #$10
DrawWarpGateInactive
        LDY xPosSecondLevelSurfaceInactivePlanet,X
        LDA secondLevelSurfaceDataInactivePlanet,X
        CLC
        ADC #$40
        STA SCREEN_RAM + LINE12_COL4,Y
        DEX
        BNE DrawWarpGateInactive
        RTS

; The *-$01 is because the array index starts at 1 rather than 0.
xPosSecondLevelSurfaceInactivePlanet =*-$01
        .BYTE $00,$01,$02,$03,$28,$29,$2A,$2B
        .BYTE $50,$51,$52,$53,$78,$79,$7A,$7B
secondLevelSurfaceDataInactivePlanet =*-$01
        .BYTE $30,$32,$38,$3A,$31,$33,$39,$3B
        .BYTE $34,$36,$3C,$3E,$35,$37,$3D,$3F
surfaceDataInactiveLowerPlanet =*-$01
        .BYTE $01,$03,$01,$03,$01,$03,$01,$03
        .BYTE $01,$03,$01,$03,$01,$03,$01,$03
        .BYTE $01,$03,$01,$03,$01,$03,$01,$03
        .BYTE $01,$03,$01,$03,$01,$03,$1D,$1F
        .BYTE $00,$02,$00,$02,$00,$02,$00,$02

textForInactiveLowerPlanet
       .TEXT "   WARP GATE        GILBY    CORE   NOT-CORE"
progressDisplaySelected .BYTE $00
```

Listing 1.17: Draw the inactive lower planet.

Enemies and their Discontents

> ACONT
>
> This is the bit that I knew would take me ages to write and get glitch free, and the bit that is absolutely necessary to the functioning of the game. The module ACONT is essentially an interpreter for my own 'wave language', allowing me to describe, exactly, an attack wave in about 50 bytes of data. The waves for the first part of IRIDIS are in good rollicking shoot-'em-up style, and there have to be plenty of them. There are five planets and each planet is to have twenty levels associated with it. It's impractical to write separate bits of code for each wave; even with 64K you can run outta memory pretty fast that way, and it's not really necessary coz a lot of stuff would be duplicated. Hence ACONT.
>
> — Jeff Minter's Development Diary in Zzap Magazine[?]

The bits and bytes that define the behaviour and appearance of wave after wave of Iridis Alpha's enemy formations - twenty across each of the five planets giving on hundred in all - takes up relatively little space given the sheer variety of adversaries the player faces.

2.0.1 You're a Waste of Space

Each 'wave' of enemies is defined by a 40 byte data structure, not 50 bytes as Minter initially suggested in his development diary. There's a little bit of waste going on in here too, bytes 10 to 14 are unused, while `Byte 15` is only ever set (to `$01`) by the wave data structure that describes the default explosion behaviour for enemy ships.

As we can see here, the sole purpose of `Byte 15` is to determine whether a new set of wave data needs to be loaded. This makes sense, once the animation is finished we'll need to load a new enemy ship. Still, you can't help thinking there might have been a way that didn't waste 99 bytes!

```
        BEQ b4D98
        LDA #$00
        STA upperPlanetAttackShipYPosUpdated ,X
        ; The 2nd stage of wave data for this enemy.
        LDY #$19
        LDA (currentShipWaveDataLoPtr) ,Y
        BEQ b4D98
        DEY
        JMP UpdateWaveDataPointersForCurrentEnemy

b4D98   LDA upperPlanetAttackShipYPosUpdated2 ,X
        BEQ No3rdWaveData
        LDA #$00
        STA upperPlanetAttackShipYPosUpdated2 ,X
        ; The 3rd stage of wave data for this enemy.
        LDY #$1B
        LDA (currentShipWaveDataLoPtr) ,Y
        BEQ No3rdWaveData
        DEY
        JMP UpdateWaveDataPointersForCurrentEnemy
```

Listing 2.1: "Routine for Animating Enemy Sprites"

And actually, it's more than that because as we shall see the `ACONT` 40-byte data structure is defined more than once per wave. Separate instances are defined for later phases of the enemy ship, such as when it is first hit. Early examples of this in the game are the 'spinning rings' you get when you hit an enemy in the first level.

In all there are 200 instances of the `ACONT` data structure: 100 defining each of the enemy waves and another 100 defining the subsequent behaviour of the ships when hit. There isn't a one-to-one mapping here either - many of the effects are reused across levels and as we shall see there can be multiple stages in an enemy's lifecycle.

So there's already 1000 bytes or 1 kilobyte of wasted space in the level data due to bytes that are never or rarely used. That's out of a total of 8 kilobytes actually used.

Shocking stuff. Awful.

2.0.2 And You're a Waste of Space

`Bytes 33 - 34` seem to be left in an unfinished state. Wave 12 on Planet 5 has both populated, `flowchartArrowAsExplosion` is the only other wave that has anything in either byte, in this case `$60` in `Byte 33`.

This another 200 wasted bytes it seems but it seems these bytes have some game logic attached and when you look at what that logic is doing it seems broken.

```
EnergyUpdateTopPlanet
        LDA extraAmountToDecreaseEnergyByTopPlanet
        BNE b4D7F
        ; Y is still $23.
        LDA (currentShipWaveDataLoPtr),Y
        JSR AugmentAmountToDecreaseEnergyByBountiesEarned
        STA extraAmountToDecreaseEnergyByTopPlanet
b4D7F   LDY #$1E
        JMP UpdateWaveDataPointersForCurrentEnemy
        ; Returns

;-----------------------------------------------------------------
; GetNewShipDataFromDataStore
```

Listing 2.2: "Routine for Animating Enemy Sprites"

When Byte 34 (`$21`) is populated (and fire has not been pressed) the game will attempt to load a set of wave date from Bytes 33 and 34:

```
        LDA #$00
        STA upperPlanetAttackShipYPosUpdated2,X
        ; The 3rd stage of wave data for this enemy.
        LDY #$1B
        LDA (currentShipWaveDataLoPtr),Y
        BEQ No3rdWaveData
        DEY
        JMP UpdateWaveDataPointersForCurrentEnemy

No3rdWaveData
        LDA joystickInput
        AND #$10
        BNE Byte34IsZero
        ; Check if we should load extra stage data for this enemy.
        ; FIXME: When this is set it would incorrectly expect there
        ; to be a hi/lo ptr in $20 and $21, when there isn't.
        LDY #$21
        LDA (currentShipWaveDataLoPtr),Y
        BEQ Byte34IsZero
```

Listing 2.3: "Routine for Animating Enemy Sprites"

In the case of the data for Planet 5 Level 12 this translates to whatever is at `$1488`. As it happens this is the address of the frequency data used in the title screen's music. So effectively pretty random data:

```
; This is the frequency table containing all the 'notes' from
; octaves 4 to 8. It's very similar to:
;  http://codebase.c64.org/doku.php?id=base:ntsc_frequency_table
; The 16 bit value you get from feeding the lo and hi bytes into
; the SID registers (see PlayNoteVoice1 and PlayNoteVoice2) plays
; the appropriate note. Each 16 bit value is based off a choice of
; based frequency. This is usually 440hz, but not here.

                    ;      C   C#  D   D#  E   F   F#  G   G#  A   A#  B
titleMusicHiBytes   .BYTE $08,$08,$09,$09,$0A,$0B,$0B,$0C,$0D,$0E,$0E,$0F  ; 4
                    .BYTE $10,$11,$12,$13,$15,$16,$17,$19,$1A,$1C,$1D,$1F  ; 5
                    .BYTE $21,$23,$25,$27,$2A,$2C,$2F,$32,$35,$38,$3B,$3F  ; 6
                    .BYTE $43,$47,$4B,$4F,$54,$59,$5E,$64,$6A,$70,$77,$7E  ; 7
                    .BYTE $86,$8E,$96,$9F,$A8,$B3,$BD,$C8,$D4,$E1,$EE,$FD  ; 8
```

Listing 2.4: "Routine for Animating Enemy Sprites"

Clearly, no one has ever reached level 12 in planet 5!

2.0.3 Clever Business

> You pass the interpreter data, that describes exactly stuff like: what each alien looks like, how many frames of animation it uses, speed of that animation, colour, velocities in X— and Y— directions, accelerations in X and Y, whether the alien should 'home in' on a target, and if so, what to home in on; whether an alien is subject to gravity, and if so, how strong is the gravity; what the alien should do if it hits top of screen, the ground, one of your bullets, or you; whether the alien can fire bullets, and if so, how frequently, and what types; how many points you get if you shoot it, and how much damage it does if it hits you; and a whole bunch more stuff like that. As you can imagine it was a fairly heavy routine to write and get debugged, but that's done now; took me about three weeks in all I'd say.
>
> — Jeff Minter's Development Diary in Zzap Magazine[?]

The level data does actually define some of this stuff. It does so by making heavy use of a simple but clever trick that in its way is very specific to 8-bit assembly programming: storing references to other data structures as a pair of bytes. We've discussed the way this works previously but we'll try again briefly here as it won't do any harm.

The 40-byte data structure that defines the default explosion animation (and behaviour, so far as it goes) is stored at position `$18C8` while the game is running. To use this explosion data when an enemy is killed, bytes 31 and 32 of the data structure contain the values `$C8,$18`.

When an enemy is hit, the game routine responsible for figuring out what to do next with it looks at bytes 31 and 32 and loads in the data structure at the address given by combining `$18` and `$C8` as the 'new' wave data that defines how that enemy ship will now behave. Since the data structure at `$18C8` basically says: animate an explosion sprite and don't move anywhere that is exactly what the enemy ship now does.

Here's the explosion data structure, which we've labelled `defaultExplosion` in our disassembly, in the first twenty bytes or so of it's gory detail:

```
defaultExplosion = $18C8
        ; Byte 1 (Index $00): An index into colorsForAttackShips that applies a
        ; color value for the ship sprite.
        .BYTE $07
        ; Byte 2 (Index $01): Sprite value for the attack ship for the upper planet.
        ; Byte 3 (Index $02): The 'end' sprite value for the attack ship's animation
        ; for the upper planet.
        .BYTE EXPLOSION_START,EXPLOSION_START + $03
        ; Byte 4 (Index $03): The animation frame rate for the attack ship.
        .BYTE $03
        ; Byte 5 (Index $04): Sprite value for the attack ship for the lower planet.
        ; Byte 6 (Index $05): The 'end' sprite value for the ship's lower planet animation.
```

```
        .BYTE EXPLOSION_START,EXPLOSION_START + $03
        ; Byte 7 (Index $06): Whether a specific attack behaviour is used.
        .BYTE $00
        ; Byte 8 (Index $07): Lo Ptr for an unused attack behaviour
        ; Byte 9 (Index $08): Hi Ptr for an unused attack behaviour
        .BYTE <nullPtr,>nullPtr
        ; Byte 10 (Index $09): Lo Ptr for an animation effect? (Doesn't seem to be used?)
        ; Byte 11 (Index $0A): Hi Ptr for an animation effect (Doesn't seem to be used?)?
        .BYTE <nullPtr,>nullPtr
        ; Byte 12 (Index $0B): some kind of rate limiting for attack wave
        .BYTE $00
        ; Byte 13 (Index $0C): Lo Ptr for a stage in wave data (never used).
        ; Byte 14 (Index $0D): Hi Ptr for a stage in wave data (never used).
        .BYTE <nullPtr,>nullPtr
        ; Byte 15 (Index $0E): Controls the rate at which new enemies are added?
        .BYTE $01
        ; Byte 16 (Index $0F): Update rate for attack wave
        .BYTE $0D
        ; Byte 17 (Index $10): Lo Ptr to the wave data we switch to when first hit.
        ; Byte 18 (Index $11): Hi Ptr to the wave data we switch to when first hit.
        .BYTE <nullPtr,>nullPtr
        ; Byte 19 (Index $12): X Pos movement for attack ship.
        .BYTE $80
        ; Byte 20 (Index $13): Y Pos movement pattern for attack ship.
```

Listing 2.5: ”Routine for Animating Enemy Sprites”

We can see the first 7 bytes are concerned with the appearance and basic behaviour of the enemy. Bytes 2 and 3 define the sprite used for display on the upper planet, Bytes 5 and 6 for the lower planet. The reason there's two in each case is because they are describing the start and end point of the sprite's animation. The game will display `EXPLOSION_START $ED`) first, then cycle through the next two sprites until it reaches `EXPLOSION_START` + 3 (`$F0`).

2.0.4 Sprites Behaving Badly

We can see this in action in `AnimateAttackShipSprites`. When this routine runs Byte 4 has been loaded to `upperPlanetAttackShipInitialFrameRate` for the upper planet and `lowerPlanetAttackShipInitialFrameRate` for the lower planet. This routine is cycling through the sprites given by Byte 2 as the lower limit and Byte 3 as the upper limit. This is what the animation consists of: an animation effect achieved by changing the sprite from one to another to create a classic animation effect.

```
upperPlanetInitialXPosFrameRateForAttackShip  .BYTE $01,$01,$01,$01
lowerPlanetInitialXPosFrameRateForAttackShip  .BYTE $01,$01,$01,$01
upperPlanetXPosFrameRateForAttackShip         .BYTE $01,$01,$01,$01
lowerPlanetXPosFrameRateForAttackShip         .BYTE $01,$01,$01,$01
upperPlanetInitialYPosFrameRateForAttackShips .BYTE $01,$01,$01,$01
lowerPlanetInitialYPosFrameRateForAttackShips .BYTE $01,$01,$01,$01
upperPlanetYPosFrameRateForAttackShips        .BYTE $01,$01,$01,$01
lowerPlanetYPosFrameRateForAttackShips        .BYTE $01,$01,$01,$01

;------------------------------------------------------------------
; UpdateAttackShipsXAndYPositions
;------------------------------------------------------------------
UpdateAttackShipsXAndYPositions
        DEC upperPlanetYPosFrameRateForAttackShips - $01,X
        BNE b7D79
        LDA upperPlanetInitialYPosFrameRateForAttackShips - $01,X
        STA upperPlanetYPosFrameRateForAttackShips - $01,X
        LDA yPosMovementForUpperPlanetAttackShips - $01,X
        CLC
        ADC upperPlanetAttackShip2YPos,X
        STA upperPlanetAttackShip2YPos,X
```

```
        AND #$F0
        CMP #$70
        BEQ b7D6A
        CMP #$00
        BNE b7D79
        LDA #$10
        STA upperPlanetAttackShip2YPos ,X
        LDA #$01
        STA upperPlanetAttackShipYPosUpdated + $01,X
        BNE b7D74
b7D6A   LDA #$6D
        STA upperPlanetAttackShip2YPos ,X
        LDA #$01
        STA upperPlanetAttackShipYPosUpdated2 + $01,X
b7D74   LDA #$00
```

Listing 2.6: ”Routine for Animating Enemy Sprites”

Byte 2 (loaded to `upperPlanetAttackShipAnimationFrameRate` comes into play here. It’s decremented and as long as it’s not zero yet the animation is skipped, execution jumps forward to `AnimateLowerPlanetAttackShips`:

```
UpdateAttackShipsXAndYPositions
        DEC upperPlanetYPosFrameRateForAttackShips - $01,X
        BNE b7D79
```

Listing 2.7: ”Routine for Animating Enemy Sprites”

If it is zero, it instead gets reset to the initial value from Byte 2 (stored in `upperPlanetAttackShipInitialFra` and the current sprite for the enemy ship is incremented to point to the next ’frame’ of the ship’s animation:

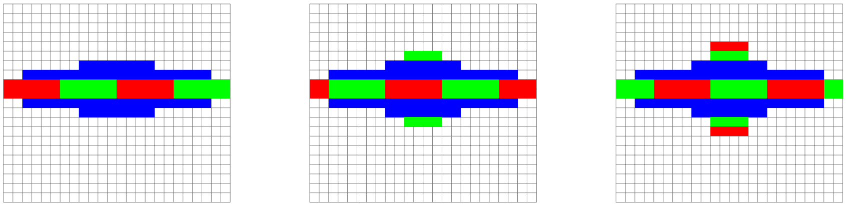
```
        STA upperPlanetYPosFrameRateForAttackShips - $01,X
        LDA yPosMovementForUpperPlanetAttackShips - $01,X
        CLC
        ADC upperPlanetAttackShip2YPos ,X
```

Listing 2.8: ”Routine for Animating Enemy Sprites”

Next we check if we’ve reached the end of the animation by checking the value of Byte 3 (loaded to `upperPlanetAttackShipSpriteAnimationEnd`). If so, we reset `upperPlanetAttackShip2SpriteValue` to the value initially loaded from Byte 2 - and that is what will be used to display the ship the next time we pass through to animate the ship:

```
        AND #$F0
        CMP #$70
        BEQ b7D6A
        CMP #$00
        BNE b7D79
        LDA #$10
        STA upperPlanetAttackShip2YPos ,X
        LDA #$01
```

Listing 2.9: ”Routine for Animating Enemy Sprites”



(a) FLYING_SAUCER1 **(b)** FLYING_SAUCER2 **(c)** FLYING_SAUCER3

Figure 2.1: The sprites used to animate the 'UFO' in the first level.

2.0.5 Enemy Movement

Enemy movement is controlled by two parameters in each direction: the number of pixels to move in one go and the number of cycles to wait between each movement. So for movement in the horizontal (or X direction) `Byte 19` controls the number of pixels to move at once, while `Byte 21` controls the number of cycles to wait between each movement. The same applies to `Byte 20` and `Byte 22` for the vertical (or Y direction).

If we look at Byte 19 and Byte 21 for Level 1 we can see that the the fast lateral movement of the 'UFO's is implemented by a relatively high value of $06 for the number of pixels it moves at each step while the interval between steps is relatively low ($01). Meanwhile the more gradual up and down movement is implemented by a value of $01 in Byte 20 and Byte 22.

For the second level ('bouncing rings') the horizontal movement is more constrained (Byte 19 is $00) while the vertical movement is more extreme (Byte 20 is $24) - achieving the bouncing effect.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$06	$01	$01	$01
2	$00	$00	$24	$02	$01
3	$00	$FA	$01	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Movement data for the first three levels.

The horizontal movement for level three is $FA, which would make you think the enemies must be moving horizontally extremely quickly. In fact, when the high bit is set a special behaviour is invoked:

```
        JSR UpdateAttackShipsXAndYPositions
b7CE9   JSR AnimateAttackShipSprites
```

Listing 2.10: "From `UpdateAttackShipsXAndYPositions.` "

When the upper bit is set (e.g. $FC,$80) on the value loaded to the accumulator by `LDA` then `BMI` will return true and jump to `UpperBitSetOnXPosMovement`.

```
        PLA
        SEC
        SBC attackShipOffsetRate
        STA upperPlanetAttackShip2XPos,X
        BCS b7CC2
        JMP j7CB9

b7D07   PHA
        TYA
        EOR #$FF
        STA attackShipOffsetRate
        INC attackShipOffsetRate
        PLA
        CLC
        ADC attackShipOffsetRate
```

Listing 2.11: "From `UpdateAttackShipsXAndYPositions.` "

This first line `EOR #$FF` performs an exclusive-or between Byte 19 in the `Accumulator` ($FA) and the value $FF. An exclusive-or, remember, is a bit by bit comparison of two bytes which will set a bit in the result if an only if the bit in one of the values is set but the other is not:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$FF	1	1	1	1	1	1	1	1
$FA	1	1	1	1	1	0	1	0
Result	0	0	0	0	0	1	0	1

X-OR'ing $FF and $FA gives $05.

This result is stored in `attackShipOffsetRate`:

```
        JMP  j7CB9
```

Listing 2.12: "From `UpdateAttackShipsXAndYPositions.` "

Incremented:

```
```

Listing 2.13: "From `UpdateAttackShipsXAndYPositions.` "

And then subtracted from the enemy's X position:

```
b7D07   PHA
        TYA
        EOR  #$FF
        STA  attackShipOffsetRate
```

Listing 2.14: "From `UpdateAttackShipsXAndYPositions.` "

The net result is a deceleration effect. This is observed in the way the licker ship wave will accelerate out to the center before dialling back again.

What is going on with Byte 7?

Byte 7 comes into play when setting the initial Y position of a new enemy. This initial vertical position is random, but subject to some adjustment:

```
SetInitialRandomPositionsOfEnemy
        LDY previousAttackShipIndex
        LDA indexForActiveShipsWaveData,X
        TAX
        LDA attackShipsMSBXPosOffsetArray + $01,X
        STA upperPlanetAttackShipsMSBXPosArray + $01,Y
        JSR PutRandomByteInAccumulatorRegister
        AND #$7F
        CLC
        ADC #$20
        STA upperPlanetAttackShipsXPosArray + $01,Y

        ; Are we on the upper or lower planet?
        TYA
        AND #$08
        BNE SetInitialRandomPositionLowerPlanet

SetInitialRandomPositionUpperPlanet
        JSR PutRandomByteInAccumulatorRegister
        AND #$3F
        CLC
        ADC #$40
        STA upperPlanetAttackShipsYPosArray + $01,Y

        STY previousAttackShipIndexTmp
        ; Byte 7 ($06): Usually an update rate for the attack
    ships.
        LDY #$06
        LDA (currentShipWaveDataLoPtr),Y
        BNE ReturnFromLoadingWaveDataEarly

        ; Byte 9 ($08): Default initiation Y position for the
    enemy.
        LDY #$08
        LDA (currentShipWaveDataLoPtr),Y
        BEQ ReturnFromLoadingWaveDataEarly

        LDA #$6C
        LDY previousAttackShipIndexTmp
        STA upperPlanetAttackShipsYPosArray + $01,Y

ReturnFromLoadingWaveDataEarly
        RTS

SetInitialRandomPositionLowerPlanet
        JSR PutRandomByteInAccumulatorRegister
```

```
        AND #$3F
        CLC
        ADC #$98
        STA upperPlanetAttackShipsYPosArray + $01,Y

        STY previousAttackShipIndexTmp
        ; Byte 7 ($06): Determines if the inital Y Position of
    the ship is random or uses a default.
        LDY #$06
        LDA (currentShipWaveDataLoPtr),Y
        BNE ReturnFromLoadingWaveData

        ; Byte 9 ($08): A Hi-Ptr to wave data normally but
    treated here .
        LDY #$08
        LDA (currentShipWaveDataLoPtr),Y
        BEQ ReturnFromLoadingWaveData

        ; Set Y Pos to $90 if we have wave data in Bytes 8-9.
        LDA #$90
        LDY previousAttackShipIndexTmp
        STA upperPlanetAttackShipsYPosArray + $01,Y

ReturnFromLoadingWaveData
        RTS
```

Listing 2.15: "The sub-routine `SetInitialRandomPositionUpperPlanet` in `GetWaveDateForNewShip`."

The first order of business is to call `PutRandomByteInAccumulatorRegister` which gets a random value and stores it in the accumulator.

```
        JSR PutRandomByteInAccumulatorRegister
```

Since `A` can now contain anything from 0 to 255 ($00 to $FF) this needs to be adjusted to a meaningful Y-position value for the upper planet. So if we imagine `PutRandomByteInAccumulatorRegister` returned $85, we now do the following operations on it:

```
        AND #$3F
        CLC
        ADC #$40
```

First we do an `AND #$3F` with the value of $85 in `A`:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
$85	1	0	0	0	0	1	0	1
$3F	0	0	1	1	1	1	1	1
Result	0	0	0	0	0	1	0	1

AND'ing $3F and $85 gives $05.

Our result is $05. The effect of the AND'ing here is to ensure that the random number we get back is between 0 and 63 rather than 0 and 255. Next we add $40 (decimal 64) to this result:

```
        CLC
        ADC  #$40
        STA  upperPlanetAttackShipsYPosArray + $01,Y
```

This gives $45 and this is what we store as the initial Y position for the enemy.

You'll notice that the steps for `SetInitialRandomPositionLowerPlanet` are identical but with only the constant of the add value of $98 instead of $40. This is simply an additional offset to ensure that the Y position is lower on the screen for the initial position of the enemy on the lower planet.

We still haven't got into what Byte 7 is doing though. With an initial Y position determined, it looks like the intention was for Byte 7 to specify some adjustment to this value. But this looks like another bit of non-functioning game logic. If Byte 7 contains a value, the function will return early without any further adjustments. If it's zero it will then try Byte 9. If that's zero, it will return early. So the logic needs Byte 7 to be zero and Byte 9 to contain something for anything to happen. That's never the case, so the the adjustment never happens:

```
        ; Byte 7 ($06): Usually an update rate for the attack
    ships.
        LDY  #$06
        LDA  (currentShipWaveDataLoPtr),Y
        BNE  ReturnFromLoadingWaveDataEarly

        ; Byte 9 ($08): Default initiation Y position for the
    enemy.
        LDY  #$08
        LDA  (currentShipWaveDataLoPtr),Y
        BEQ  ReturnFromLoadingWaveDataEarly
```

```
        LDA #$6C
        LDY previousAttackShipIndexTmp
        STA upperPlanetAttackShipsYPosArray + $01,Y
```

Listing 2.16: "An adjustment that never happens. Byte 7 and Byte 9 are never set in this way"

This is definitely some forgotten code. Byte 7 is elsewhere used in combination with Byte 8 and Byte 9 to define an alternate enemy mode for some levels where the ship will supplement any dead ships with alternate enemy types and attack patterns periodically.

2.0.6 Pointer Data

This happens in `MaybeSwitchToAlternateEnemyPattern` in `UpdateAttackShipDataForNewShip`.

```
        ; Byte 7 ($06): Usually an update rate for the attack
    ships.
        LDY #$06
        LDA (currentShipWaveDataLoPtr),Y
        BEQ EarlyReturnFromAttackShipBehaviour

        DEC rateForSwitchingToAlternateEnemy,X
        BNE EarlyReturnFromAttackShipBehaviour

        LDA (currentShipWaveDataLoPtr),Y
        STA rateForSwitchingToAlternateEnemy,X

        ; Push the current ship's position data onto the stack.
        TXA
        PHA
        LDY indexIntoUpperPlanetAttackShipXPosAndYPosArray,X
        LDA upperPlanetAttackShipsXPosArray + $01,Y
        PHA
        LDA upperPlanetAttackShipsMSBXPosArray + $01,Y
        PHA
        LDA upperPlanetAttackShipsYPosArray + $01,Y
        PHA

        ; Are we on the top or bottom planet?
        TXA
        AND #$08
        BNE LowerPlanetAttackShipBehaviour
```

```
        ; We're on the upper planet.
        LDX #$02
ProcessAttackShipBehaviour
        JSR SetXToIndexOfShipThatNeedsReplacing
        BEQ ResetAndReturnFromAttackShipBehaviour

        ; Pop the current ship's position data from the stack
    and
        ; populate it into the new ship's position.
        LDY indexIntoUpperPlanetAttackShipXPosAndYPosArray,X
        PLA
        STA upperPlanetAttackShipsYPosArray + $01,Y
        PLA
        BEQ MSBXPosOffsetIzZero

        LDA attackShipsMSBXPosOffsetArray + $01,X
MSBXPosOffsetIzZero
        STA upperPlanetAttackShipsMSBXPosArray + $01,Y
        PLA
        STA upperPlanetAttackShipsXPosArray + $01,Y
        PLA

        ; Byte 8 of Wave Data gets loaded now. Bytes 8 and 9
```

Listing 2.17: "Byte 7 is used to periodically switch to an enemy mode defined by Bytes 8-9"

Byte 7 is used to drive the rate at which this routine switches over to the enemy data/-mode defined by Byte 8 and Byte 9.

```
        BNE EarlyReturnFromAttackShipBehaviour

        LDA (currentShipWaveDataLoPtr),Y
        STA rateForSwitchingToAlternateEnemy,X
```

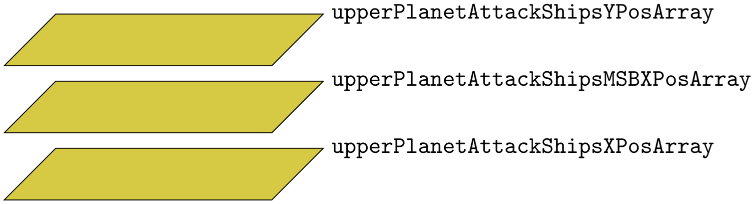
Listing 2.18: "`rateForSwitchingToAlternateEnemy` (Byte 7) is decremented and reloaded each time it reaches zero."

What this routine is going to do is replace the first dead ship it finds in the current wave with the wave data pointed to by Byte 8-9 and create a new enemy with the current ship's position with it.

First, we store the current ship's position. The way to do this is get the index (`Y`) for the current ship `X` and store each of the X and Y Position information into the accumulator first `A` and then push it onto the 'stack' (`PHA` which means 'push `A` onto the stack').

```
        TXA
        PHA
        LDY indexIntoUpperPlanetAttackShipXPosAndYPosArray,X
        LDA upperPlanetAttackShipsXPosArray + $01,Y
        PHA
        LDA upperPlanetAttackShipsMSBXPosArray + $01,Y
        PHA
        LDA upperPlanetAttackShipsYPosArray + $01,Y
        PHA
```

When this has run the stack of accumulator values now looks like this:



The stack after the code above has run with `upperPlanetAttackShipsXPosArray` at the top.

With our position data safely stashed away on the stack we now decide which planet we're on:

```
        TXA
        AND #$08
        BNE LowerPlanetAttackShipBehaviour
```

If we're on the upper planet we use `SetXToIndexOfShipThatNeedsReplacing` look in the `activeShipsWaveDataHiPtrArray` for any ships that need replacing between positions `$02` and `$06`. If we don't find one, we return early:

```
        LDX #$02
ProcessAttackShipBehaviour
        JSR SetXToIndexOfShipThatNeedsReplacing
        BEQ ResetAndReturnFromAttackShipBehaviour
```

If we do find one we can now pull (or 'pop') the positional data we stored away in the stack and assign that to the once-dead ship. First we use the index we retrieved to `X` to get the ship's index (`Y`) into the positional arrays:

```
        ; populate it into the new ship's position.
```

```
        LDY  indexIntoUpperPlanetAttackShipXPosAndYPosArray,X
        PLA
```

Then we pop the first positional item `upperPlanetAttackShipsYPosArray` from the top of the stack and store in the new ship's position in the array:

```
        STA  upperPlanetAttackShipsYPosArray + $01,Y
        PLA
```

Listing 2.19: "`PLA` remove the top item from the stack and stores it in `A`

The stack now looks like this, popping from the stack has the effect of removing the first item:

Then we pop the rest of the items one by one and assign them to the new ship. We ignore the sprite's MXB offset if it is zero:

```
        BEQ  MSBXPosOffsetIzZero

        LDA  attackShipsMSBXPosOffsetArray + $01,X
MSBXPosOffsetIzZero
        STA  upperPlanetAttackShipsMSBXPosArray + $01,Y
        PLA
        STA  upperPlanetAttackShipsXPosArray + $01,Y
        PLA
```

Listing 2.20: "`PLA` remove the top item from the stack and stores it in `A`

Now that we have set up the positional data for the new enemy we load all its other features from the data pointed to by Byte 8-9:

```
        ; contain the hi/lo ptrs to the alternate enemy data.
        LDY  #$07
        JMP  UpdateWaveDataPointersForCurrentEnemy
```

Let's take a closer look at this routine `UpdateWaveDataPointersForCurrentEnemy`. What it does in this instance is take the address pointed to by Bytes 8 and 9 and load the data there using the routine `GetWaveDataForNewShip`. To be used in this way the values in Bytes 8 and 9 are combined and treated as an address in memory. For example if Byte 8 contains `$70` and Byte 9 contains `$13` they are treated as providing the address `$1370`. This is the location where the enemy data for `planet1Level8Data` is kept so that is what is loaded.

Planet	Level	Byte 7	Bytes 8-9
1	11	$03	smallDotWaveData
1	14	$03	planet1Level8Data
2	19	$0C	landGilbyAsEnemy
3	4	$04	gilbyLookingLeft
3	6	$04	planet3Level6Additional
4	19	$01	planet4Level19Additional
5	3	$01	planet5Level3Additional
5	5	$05	planet5Level5Additional
5	14	$06	llamaWaveData

Byte 7 : Whether a specific attack behaviour is used.
Bytes 8-9 : Lo and Hi Ptr for alternate enemy mode

Table 2.1: Actual use of Bytes 7, 8, and 9. Note that the value in Byte 7 doesn't matter, as long as it's non-zero.

2.0.7 Enemy Behaviour

2.0.8 Level Movement Data

Appendix: Enemy Data

Sprite Data for Each Level

Level	Byte 1	Byte 3	Byte 4	Byte 6
1	$06	FLYING_SAUCER1	$03	FLYING_SAUCER1
2	$06	BOUNCY_RING	$01	BOUNCY_RING
3	$05	FLYING_DOT1	$04	FLYING_DOT1
4	$11	FLYING_TRIANGLE1	$03	FLYING_TRIANGLE1
5	$11	BALLOON	$00	BIRD1
6	$0A	BIRD1	$03	BIRD1
7	$09	FLAG_BAR	$00	FLAG_BAR
8	$11	TEARDROP_EXPLOSION1	$03	TEARDROP_EXPLOSION1
9	$06	WINGBALL	$03	MONEY_BAG
10	$08	CAMEL	$00	INV_MAGIC_MUSHROOM
11	$0E	GILBY_AIRBORNE_LEFT	$06	GILBY_AIRBORNE_LOWERPLANET_RIGHT
12	$09	CAMEL	$02	LICKERSHIP_INV1
13	$0B	BUBBLE	$04	BUBBLE
14	$06	TEARDROP_EXPLOSION1	$05	TEARDROP_EXPLOSION1
15	$08	LLAMA	$00	LLAMA
16	$05	QBERT_SQUARES	$00	QBERT_SQUARES
17	$0A	BOUNCY_RING	$02	BOUNCY_RING
18	$05	GILBY_AIRBORNE_RIGHT	$00	GILBY_AIRBORNE_RIGHT
19	$04	STARSHIP	$00	STARSHIP
20	$07	COPTIC_CROSS	$00	COPTIC_CROSS

Byte 1 : Index into array for sprite color
Byte 3 : Sprite value for the attack ship on the upper planet
Byte 4 : The animation frame rate for the attack ship.
Byte 6 : Sprite value for the attack ship on lower planet

Planet 1 - Sprite Data.

Level	Byte 1	Byte 3	Byte 4	Byte 6
1	$55	LITTLE_DART	$01	LITTLE_DART
2	$04	FLYING_COCK1	$05	FLYING_COCK1
3	$06	FLYING_COCK_RIGHT1	$05	FLYING_COCK_RIGHT1
4	$05	TEARDROP_EXPLOSION1	$01	TEARDROP_EXPLOSION1
5	$05	LICKER_SHIP1	$00	LICKERSHIP_INV1
6	$04	SPINNING_RING1	$00	SPINNING_RING1
7	$0F	SMALLBALL_AGAIN	$00	SMALLBALL_AGAIN
8	$0C	BUBBLE	$04	BUBBLE
9	$04	LAND_GILBY1	$03	LAND_GILBY_LOWERPLANET1
10	$11	FLYING_TRIANGLE1	$00	FLYING_TRIANGLE1
11	$00	FLYING_SAUCER1	$01	FLYING_SAUCER1
12	$0C	BUBBLE	$01	BUBBLE
13	$08	LLAMA	$00	LLAMA
14	$04	FLYING_COCK1	$05	FLYING_COCK1
15	$08	FLAG_BAR	$00	FLAG_BAR
16	$10	WINGBALL	$04	MONEY_BAG
17	$06	FLYING_COCK_RIGHT1	$05	FLYING_COCK_RIGHT1
18	$10	BOLAS1	$02	BOLAS1
19	$0E	LAND_GILBY1	$04	LAND_GILBY_LOWERPLANET1
20	$06	EYE_OF_HORUS	$00	EYE_OF_HORUS

Byte 1 : Index into array for sprite color
Byte 3 : Sprite value for the attack ship on the upper planet
Byte 4 : The animation frame rate for the attack ship.
Byte 6 : Sprite value for the attack ship on lower planet

Planet 2 - Sprite Data.

Level	Byte 1	Byte 3	Byte 4	Byte 6
1	$10	$FC	$02	$FC
2	$0D	LITTLE_DART	$00	LITTLE_DART
3	$02	BOUNCY_RING	$04	BOUNCY_RING
4	$06	GILBY_AIRBORNE_RIGHT	$00	GILBY_AIRBORNE_RIGHT
5	$0B	SMALL_BALL1	$02	SMALL_BALL1
6	$00	LAND_GILBY1	$01	LAND_GILBY_LOWERPLANET1
7	$07	LICKER_SHIP1	$07	LICKERSHIP_INV1
8	$0C	BUBBLE	$03	BUBBLE
9	$06	FLYING_DART1	$05	FLYING_DART1
10	$06	FLYING_SAUCER1	$03	FLYING_SAUCER1
11	$04	LICKER_SHIP1	$05	LICKERSHIP_INV1
12	$00	SMALLBALL_AGAIN	$00	SMALLBALL_AGAIN
13	$06	LICKER_SHIP1	$05	LICKERSHIP_INV1
14	$08	CAMEL	$00	CAMEL
15	$10	BOUNCY_RING	$01	BOUNCY_RING
16	$10	STRANGE_SYMBOL	$00	STRANGE_SYMBOL
17	$08	LLAMA	$00	LLAMA
18	$06	FLYING_SAUCER1	$01	FLYING_SAUCER1
19	$0E	FLYING_COMMA1	$04	FLYING_COMMA1
20	$11	PSI	$00	PSI

Byte 1 : Index into array for sprite color
Byte 3 : Sprite value for the attack ship on the upper planet
Byte 4 : The animation frame rate for the attack ship.
Byte 6 : Sprite value for the attack ship on lower planet

Planet 3 - Sprite Data.

Level	Byte 1	Byte 3	Byte 4	Byte 6
1	$04	MAGIC_MUSHROOM	$00	INV_MAGIC_MUSHROOM
2	$0E	GILBY_AIRBORNE_RIGHT	$00	GILBY_AIRBORNE_RIGHT
3	$02	LITTLE_DART	$03	LITTLE_DART
4	$03	MAGIC_MUSHROOM	$00	INV_MAGIC_MUSHROOM
5	$09	LOZENGE	$00	LOZENGE
6	$06	SMALLBALL_AGAIN	$00	SMALLBALL_AGAIN
7	$05	TEARDROP_EXPLOSION1	$06	TEARDROP_EXPLOSION1
8	$00	LLAMA	$00	LLAMA
9	$11	BUBBLE	$04	BUBBLE
10	$11	FLYING_COCK_RIGHT1	$05	FLYING_COCK_RIGHT1
11	$0E	MAGIC_MUSHROOM	$00	INV_MAGIC_MUSHROOM
12	$00	SMALLBALL_AGAIN	$00	SMALLBALL_AGAIN
13	$0D	FLYING_DOT1	$03	FLYING_DOT1
14	$11	SMALLBALL_AGAIN	$00	SMALLBALL_AGAIN
15	$10	BOLAS1	$02	BOLAS1
16	$07	CAMEL	$00	CAMEL
17	$00	CUMMING_COCK1	$06	BOLAS1
18	$10	CUMMING_COCK1	$05	LICKERSHIP_INV1
19	$06	QBERT_SQUARES	$00	QBERT
20	$10	BULLHEAD	$00	BULLHEAD

Byte 1 : Index into array for sprite color
Byte 3 : Sprite value for the attack ship on the upper planet
Byte 4 : The animation frame rate for the attack ship.
Byte 6 : Sprite value for the attack ship on lower planet

Planet 4 - Sprite Data.

Level	Byte 1	Byte 3	Byte 4	Byte 6
1	$11	STARSHIP	$00	STARSHIP
2	$11	MAGIC_MUSHROOM	$00	INV_MAGIC_MUSHROOM
3	$02	LAND_GILBY1	$04	LAND_GILBY_LOWERPLANET1
4	$0E	TEARDROP_EXPLOSION1	$04	TEARDROP_EXPLOSION1
5	$04	FLYING_COMMA1	$05	FLYING_COMMA1
6	$0B	STARSHIP	$00	STARSHIP
7	$10	FLYING_FLOWCHART1	$01	FLYING_FLOWCHART1
8	$10	BALLOON	$01	BOUNCY_RING
9	$00	BOUNCY_RING	$03	BOUNCY_RING
10	$08	CAMEL	$00	CAMEL
11	$06	BIRD1	$04	BIRD1
12	$07	BALLOON	$03	LAND_GILBY_LOWERPLANET8
13	$11	BUBBLE	$01	BUBBLE
14	$08	CAMEL	$00	CAMEL
15	$10	BOUNCY_RING	$04	BOUNCY_RING
17	$10	BUBBLE	$02	BUBBLE
18	$10	LITTLE_OTHER_EYEBALL	$01	SMALL_BALL1
20	$02	ATARI_ST	$00	ATARI_ST

Byte 1 : Index into array for sprite color
Byte 3 : Sprite value for the attack ship on the upper planet
Byte 4 : The animation frame rate for the attack ship.
Byte 6 : Sprite value for the attack ship on lower planet

Planet 5 - Sprite Data.

3.0.1 Enemy Pointer Data

Level	Byte 9	Byte 18	Byte 26	Byte 28	Byte 30	Byte 32
1	nullPtr	planet1Level1Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
2	nullPtr	nullPtr	nullPtr	planet1Level2Data	spinningRings	planet1Level2Data
3	nullPtr	planet1Level3Data2ndStage	nullPtr	nullPtr	lickerShipWaveData	lickerShipWaveData
4	nullPtr	planet1Level4Data2ndStage	nullPtr	nullPtr	planet1Level4Data2ndStage	planet1Level4Data2ndStage
5	nullPtr	nullPtr	nullPtr	planet1Level5Data2ndStage	planet1Level5Data3rdStage	defaultExplosion
6	nullPtr	planet1Level6Data2ndStage	nullPtr	nullPtr	spinningRings2ndType	defaultExplosion
7	nullPtr	planet1Level7Data2ndStage	nullPtr	nullPtr	planet1Level7Data2ndStage	defaultExplosion
8	nullPtr	nullPtr	nullPtr	nullPtr	planet1Level8Data2ndStage	planet1Level8Data2ndStage
9	nullPtr	planet1Level9DataSecondStage	planet1Level9DataSecondStage	nullPtr	defaultExplosion	defaultExplosion
10	nullPtr	planet1Level10Data2ndStage	nullPtr	planet1Level10Data	planet1Level10Data2ndStage	defaultExplosion
11	smallDotWaveData	nullPtr	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
12	nullPtr	nullPtr	nullPtr	planet1Level12Data	planet1Level2Data2ndStage	defaultExplosion
13	nullPtr	nullPtr	nullPtr	planet1Level13Data	planet1Level13Data2ndStage	planet1Level13Data2ndStage
14	planet1Level8Data	nullPtr	nullPtr	nullPtr	planet1Level8Data	planet1Level8Data
15	nullPtr	planet1Level15Data	nullPtr	nullPtr	teardropExplosion	lickerShipWaveData
16	nullPtr	nullPtr	nullPtr	nullPtr	planet4Level19Data	defaultExplosion
17	nullPtr	planet1Level17Data2ndStage	nullPtr	nullPtr	gilbyLookingLeft	defaultExplosion
18	nullPtr	nullPtr	nullPtr	nullPtr	planet1Level18Data2ndStage	defaultExplosion
19	nullPtr	planet1Level19Data	nullPtr	nullPtr	planet5Level6Data	planet5Level6Data
20	nullPtr	nullPtr	copticExplosion	nullPtr	planet1Level20Data	planet1Level20Data

Byte 9 : Hi Ptr for an unused attack behaviour
Byte 18: Hi Ptr to the wave data we switch to when first hit.
Byte 26: Hi Ptr for another set of wave data.
Byte 28: Hi Ptr for another set of wave data.
Byte 30: Hi Ptr for Explosion animation.
Byte 32: Hi Ptr for another set of wave data for this level.

Planet 1 - Pointer Data.

Level	Byte 9	Byte 18	Byte 26	Byte 28	Byte 30	Byte 32
1	nullPtr	planet2Level1Data	nullPtr	nullPtr	pinAsExplosion	defaultExplosion
2	nullPtr	nullPtr	nullPtr	nullPtr	secondExplosionAnimation	lickerShipWaveData
3	nullPtr	nullPtr	nullPtr	nullPtr	secondExplosionAnimation	lickerShipWaveData
4	nullPtr	nullPtr	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
5	nullPtr	nullPtr	nullPtr	planet2Level5Data2ndStage	planet2Level5Data3rdStage	planet2Level5Data2ndStage
6	nullPtr	nullPtr	planet2Level6Data2ndStage	nullPtr	secondExplosionAnimation	lickerShipWaveData
7	nullPtr	planet2Level7Data2ndStage	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
8	nullPtr	planet2Level8Data2ndStage	nullPtr	nullPtr	planet2Level8Data2ndStage	defaultExplosion
9	nullPtr	nullPtr	nullPtr	planet2Level9Data	gilbyTakingOffAsExplosion	defaultExplosion
10	nullPtr	nullPtr	nullPtr	nullPtr	flowchartArrowAsExplosion	defaultExplosion
11	nullPtr	nullPtr	nullPtr	nullPtr	nullPtr	planet2Level11Data2ndStage
12	nullPtr	nullPtr	nullPtr	nullPtr	planet2Level1Data	defaultExplosion
13	nullPtr	nullPtr	nullPtr	nullPtr	planet2Level13Data2ndStage	defaultExplosion
14	nullPtr	nullPtr	nullPtr	nullPtr	planet2Level14Data2ndStage	lickerShipWaveData
15	nullPtr	nullPtr	nullPtr	planet2Level15Data	planet2Level15Data2ndStage	defaultExplosion
16	nullPtr	nullPtr	nullPtr	nullPtr	planet1Level9Data	defaultExplosion
17	nullPtr	nullPtr	nullPtr	nullPtr	planet2Level17Data2ndStage	lickerShipWaveData
18	nullPtr	planet2Level18Data2ndStage	nullPtr	nullPtr	defaultExplosion	defaultExplosion
19	landGilbyAsEnemy	nullPtr	nullPtr	planet2Level19Data	planet2Level19Data2ndStage	defaultExplosion
20	nullPtr	nullPtr	copticExplosion	nullPtr	planet2Level20Data	planet2Level20Data

Byte 9 : Hi Ptr for an unused attack behaviour
Byte 18: Hi Ptr to the wave data we switch to when first hit.
Byte 26: Hi Ptr for another set of wave data.
Byte 28: Hi Ptr for another set of wave data.
Byte 30: Hi Ptr for Explosion animation.
Byte 32: Hi Ptr for another set of wave data for this level.

Planet 2 - Pointer Data.

Level	Byte 9	Byte 18	Byte 26	Byte 28	Byte 30	Byte 32
1	nullPtr	planet3Level1Data	nullPtr	nullPtr	$50	defaultExplosion
2	nullPtr	planet3Level2Data2ndStage	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
3	nullPtr	nullPtr	nullPtr	planet3Level3Data2ndStage	secondExplosionAnimation	defaultExplosion
4	gilbyLookingLeft	nullPtr	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
5	nullPtr	nullPtr	nullPtr	nullPtr	stickyGlobeExplosion	planet3Level5Data
6	planet3Level6Additional	nullPtr	nullPtr	planet3Level6Data	planet2Level9Data	defaultExplosion
7	nullPtr	nullPtr	planet3Level7Data2ndStage	nullPtr	spinningRings	defaultExplosion
8	nullPtr	nullPtr	nullPtr	nullPtr	bubbleExplosion	defaultExplosion
9	nullPtr	planet3Level9Data2ndStage	nullPtr	nullPtr	secondExplosionAnimation	defaultExplosion
10	nullPtr	planet3Level10Data2ndStage	nullPtr	nullPtr	spinningRings	planet3Level10Data
11	nullPtr	nullPtr	nullPtr	planet3Level11Data	planet3Level11Data2ndStage	defaultExplosion
12	nullPtr	nullPtr	nullPtr	planet3Level12Data	planet3Level12Data2ndStage	defaultExplosion
13	nullPtr	nullPtr	nullPtr	nullPtr	lickerShipAsExplosion	defaultExplosion
14	nullPtr	planet1Level12Data	nullPtr	nullPtr	planet3Level14Data2ndStage	defaultExplosion
15	nullPtr	nullPtr	nullPtr	nullPtr	planet3Level15Data2ndStage	defaultExplosion
16	nullPtr	planet2Level5Data	nullPtr	nullPtr	planet3Level16Data	defaultExplosion
17	nullPtr	nullPtr	nullPtr	nullPtr	planet5Level14Data	defaultExplosion
18	nullPtr	nullPtr	nullPtr	nullPtr	planet3Level18Data2ndStage	planet3Level18Data2ndStage
19	nullPtr	planet3Level19Data2ndStage	nullPtr	nullPtr	planet4Level17Data	planet4Level17Data
20	nullPtr	nullPtr	copticExplosion	nullPtr	planet3Level20Data	planet3Level20Data

Byte 9 : Hi Ptr for an unused attack behaviour
Byte 18: Hi Ptr to the wave data we switch to when first hit.
Byte 26: Hi Ptr for another set of wave data.
Byte 28: Hi Ptr for another set of wave data.
Byte 30: Hi Ptr for Explosion animation.
Byte 32: Hi Ptr for another set of wave data for this level.

Planet 3 - Pointer Data.

Level	Byte 9	Byte 18	Byte 26	Byte 28	Byte 30	Byte 32
1	nullPtr	nullPtr	nullPtr	planet4Level1Data2ndStage	spinningRings	defaultExplosion
2	nullPtr	nullPtr	nullPtr	nullPtr	planet4Leve2Data2ndStage	defaultExplosion
3	nullPtr	planet4Level2Data2ndStage	nullPtr	nullPtr	planet1Level5Data3rdStage	defaultExplosion
4	nullPtr	planet4Level4Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
5	nullPtr	nullPtr	nullPtr	planet4Level5Data2ndStage	secondExplosionAnimation	defaultExplosion
6	nullPtr	planet4Level6Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
7	nullPtr	planet1Level14Data	nullPtr	nullPtr	defaultExplosion	defaultExplosion
8	nullPtr	nullPtr	nullPtr	nullPtr	planet4Level8Data	planet4Level8Data2ndStage
9	nullPtr	nullPtr	nullPtr	planet4Level9Data2ndStage	spinningRings	lickerShipWaveData
10	nullPtr	planet4Level10Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
11	nullPtr	planet4Level11Data2ndStage	nullPtr	nullPtr	planet4Level11Data2ndStage	planet4Level11Data2ndStage
12	nullPtr	nullPtr	nullPtr	nullPtr	planet4Level12Data2ndStage	defaultExplosion
13	nullPtr	nullPtr	nullPtr	nullPtr	planet5Level5Data	planet5Level5Data
14	nullPtr	nullPtr	nullPtr	nullPtr	planet4Level14Data2ndStage	defaultExplosion
15	nullPtr	nullPtr	nullPtr	nullPtr	spinnerAsExplosion	defaultExplosion
16	nullPtr	nullPtr	nullPtr	nullPtr	planet4Level16Data2ndStage	defaultExplosion
17	nullPtr	nullPtr	nullPtr	nullPtr	cummingCock	defaultExplosion
18	nullPtr	nullPtr	nullPtr	planet4Level18Data	secondExplosionAnimation	defaultExplosion
19	planet4Level19Additional	nullPtr	nullPtr	planet4Level19Data	spinningRings	defaultExplosion
20	nullPtr	nullPtr	copticExplosion	nullPtr	planet4Level20Data	planet4Level20Data

Byte 9 : Hi Ptr for an unused attack behaviour
Byte 18: Hi Ptr to the wave data we switch to when first hit.
Byte 26: Hi Ptr for another set of wave data.
Byte 28: Hi Ptr for another set of wave data.
Byte 30: Hi Ptr for Explosion animation.
Byte 32: Hi Ptr for another set of wave data for this level.

Planet 4 - Pointer Data.

Level	Byte 9	Byte 18	Byte 26	Byte 28	Byte 30	Byte 32
1	nullPtr	planet5Level1Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
2	nullPtr	nullPtr	nullPtr	planet5Level2Data	planet5Level2Explosion	defaultExplosion
3	planet5Level3Additional	nullPtr	nullPtr	planet5Level3Data	planet5Level3Data2ndStage	lickerShipWaveData
4	nullPtr	planet5Level5Data2ndStage	nullPtr	nullPtr	spinningRings	lickerShipWaveData
5	planet5Level5Additional	planet5Level5Data2ndStage	nullPtr	nullPtr	spinningRings	defaultExplosion
6	nullPtr	nullPtr	nullPtr	nullPtr	fighterShipAsExplosion	defaultExplosion
7	nullPtr	nullPtr	nullPtr	nullPtr	planet5Level7Data2ndStage	defaultExplosion
8	nullPtr	nullPtr	nullPtr	planet5Level8Data	planet1Level5Data	defaultExplosion
9	nullPtr	planet5Level9Data2ndStage	nullPtr	nullPtr	planet5Level9Data2ndStage	defaultExplosion
10	nullPtr	nullPtr	defaultExplosion	nullPtr	lickerShipWaveData	lickerShipWaveData
11	nullPtr	planet5Level11Data	nullPtr	nullPtr	planet5Level11Data2ndStage	defaultExplosion
12	nullPtr	nullPtr	planet1Level5Data	planet5Level12Data	nullPtr	defaultExplosion
13	nullPtr	nullPtr	nullPtr	nullPtr	planet5Level13Data2ndStage	defaultExplosion
14	llamaWaveData	nullPtr	nullPtr	nullPtr	spinningRings	lickerShipWaveData
15	nullPtr	nullPtr	nullPtr	nullPtr	planet5Level15Data2ndStage	defaultExplosion
17	nullPtr	nullPtr	nullPtr	planet5Level17Data	planet3Level8Data	defaultExplosion
18	nullPtr	planet5Level18Data	nullPtr	nullPtr	planet1Level5Data3rdStage	defaultExplosion
20	nullPtr	nullPtr	copticExplosion	nullPtr	planet5Level20Data	planet5Level20Data

Byte 9 : Hi Ptr for an unused attack behaviour
Byte 18: Hi Ptr to the wave data we switch to when first hit.
Byte 26: Hi Ptr for another set of wave data.
Byte 28: Hi Ptr for another set of wave data.
Byte 30: Hi Ptr for Explosion animation.
Byte 32: Hi Ptr for another set of wave data for this level.

Planet 5 - Pointer Data.

3.0.2 Enemy Behaviour

Level	Byte 16	Byte 23	Byte 24	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
1	$40	$00	$00	$02	$02	$00	$04	$18
2	$00	$01	$23	$01	$01	$00	$04	$20
3	$30	$00	$00	$02	$01	$00	$04	$20
4	$60	$00	$01	$04	$02	$00	$04	$20
5	$00	$00	$23	$00	$05	$00	$04	$20
6	$03	$01	$01	$01	$04	$00	$04	$10
7	$50	$00	$01	$00	$03	$00	$04	$28
8	$00	$00	$01	$02	$02	$00	$04	$20
9	$0C	$00	$00	$00	$08	$00	$04	$20
10	$80	$00	$23	$00	$06	$00	$04	$18
11	$00	$00	$01	$04	$05	$00	$04	$10
12	$00	$00	$23	$03	$02	$00	$04	$20
13	$00	$01	$23	$04	$02	$00	$04	$20
14	$00	$00	$01	$00	$08	$00	$04	$10
15	$10	$01	$01	$03	$03	$00	$04	$20
16	$00	$01	$00	$00	$06	$00	$04	$18
17	$40	$00	$00	$05	$0C	$00	$04	$20
18	$00	$00	$01	$00	$03	$00	$04	$20
19	$20	$80	$01	$00	$04	$00	$04	$20
20	$00	$00	$23	$01	$01	$00	$04	$40

Byte 16: Update rate for attack wave
Byte 23: Stickiness factor, does the enemy stick to the player
Byte 24: Does the enemy gravitate quickly toward the player when its hit?
Byte 35: Does destroying this enemy increase the gilby's energy?.
Byte 36: Does colliding with this enemy decrease the gilby's energy?
Byte 37: Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38: Number of waves in data.
Byte 39: Number of ships in wave.

Planet 1 - Enemy Behaviour Data.

Level	Byte 16	Byte 23	Byte 24	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
1	$08	$01	$10	$01	$02	$00	$04	$18
2	$00	$00	$01	$02	$02	$00	$04	$18
3	$00	$00	$01	$02	$02	$00	$04	$18
4	$00	$00	$23	$06	$03	$00	$04	$18
5	$00	$00	$23	$02	$01	$00	$04	$30
6	$00	$00	$00	$01	$02	$00	$04	$20
7	$40	$00	$01	$03	$02	$00	$04	$10
8	$60	$00	$23	$00	$20	$00	$04	$10
9	$00	$00	$23	$00	$04	$00	$04	$10
10	$00	$00	$00	$00	$06	$00	$04	$18
11	$00	$10	$01	$00	$00	$00	$04	$10
12	$00	$00	$00	$00	CAMEL	$00	$04	$30
13	$00	$01	$01	$02	$01	$00	$04	$40
14	$00	$00	$01	$02	$02	$00	$04	$18
15	$00	$00	$23	$03	$02	$00	$04	$20
16	$00	$00	$00	$00	$10	$00	$04	$30
17	$00	$00	$01	$02	$02	$00	$04	$18
18	$08	$01	$01	$00	$06	$00	$04	$20
19	$00	$00	$23	$04	$02	$00	$04	$38
20	$00	$00	$23	$01	$01	$00	$04	$40

Byte 16: Update rate for attack wave
Byte 23: Stickiness factor, does the enemy stick to the player
Byte 24: Does the enemy gravitate quickly toward the player when its hit?
Byte 35: Does destroying this enemy increase the gilby's energy?.
Byte 36: Does colliding with this enemy decrease the gilby's energy?
Byte 37: Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38: Number of waves in data.
Byte 39: Number of ships in wave.

Planet 2 - Enemy Behaviour Data.

Level	Byte 16	Byte 23	Byte 24	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
1	$20	$01	$01	$01	$01	$53	$41	$56
2	$50	$00	$00	$02	$05	$00	$04	$18
3	$00	$00	$23	$02	$04	$00	$04	$10
4	$00	$00	$01	$04	$08	$00	$04	$18
5	$00	$00	$00	$02	$03	$00	$04	$20
6	$00	$00	$23	$03	$04	$00	$04	$10
7	$00	$00	$00	$01	$02	$00	$04	$20
8	$00	$00	$00	$00	$0C	$00	$04	$10
9	$0C	$01	$01	$03	$05	$00	$04	$18
10	$0A	$01	$01	$03	$03	$00	$04	$20
11	$00	$00	$23	$00	$08	$00	$04	$20
12	$00	$00	$23	$01	$02	$00	$04	$28
13	$00	$00	$00	$00	$05	$00	$04	$18
14	$F0	$00	$00	$00	$08	$00	$04	$20
15	$00	$00	$00	$03	$02	$00	$04	$28
16	$C0	$00	$00	$00	$10	$00	$04	$10
17	$00	$00	$01	$00	$0C	$00	$04	$30
18	$00	$01	$01	$04	$02	$00	$04	$20
19	$40	$00	$01	$00	$04	$00	$04	$20
20	$00	$00	$23	$01	$01	$00	$04	$40

Byte 16: Update rate for attack wave
Byte 23: Stickiness factor, does the enemy stick to the player
Byte 24: Does the enemy gravitate quickly toward the player when its hit?
Byte 35: Does destroying this enemy increase the gilby's energy?.
Byte 36: Does colliding with this enemy decrease the gilby's energy?
Byte 37: Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38: Number of waves in data.
Byte 39: Number of ships in wave.

Planet 3 - Enemy Behaviour Data.

Level	Byte 16	Byte 23	Byte 24	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
1	$00	$00	$23	$02	$02	$00	$04	$20
2	$00	$00	$01	$04	$01	$00	$04	$10
3	$60	$00	$02	$00	$03	$00	$04	$20
4	$40	$00	$23	$02	$04	$00	$04	$18
5	$00	$00	$23	$02	$04	$00	$04	$18
6	$20	$00	$00	$01	$04	$00	$04	$20
7	$E0	$00	$00	$00	$08	$00	$04	$08
8	$00	$00	$00	$00	$00	$00	$04	$20
9	$00	$00	$23	$04	$01	$00	$04	$20
10	$10	$00	$01	$00	$08	$00	$04	$20
11	$E0	$00	$23	$02	$02	$00	$04	$20
12	$00	$00	$00	$00	$04	$00	$04	$30
13	$00	$00	$01	$00	$0C	$00	$04	$40
14	$00	$00	$00	$00	$10	$00	$04	$18
15	$00	$01	$01	$03	$03	$00	$04	$28
16	$00	$00	$01	$00	$0C	$00	$04	$30
17	$00	$00	$00	$00	$0C	$00	$04	$20
18	$00	$01	$23	$03	$02	$00	$04	$20
19	$00	$00	$23	$02	$08	$00	$04	$0C
20	$00	$00	$23	$01	$01	$05	$05	$05

Byte 16: Update rate for attack wave
Byte 23: Stickiness factor, does the enemy stick to the player
Byte 24: Does the enemy gravitate quickly toward the player when its hit?
Byte 35: Does destroying this enemy increase the gilby's energy?.
Byte 36: Does colliding with this enemy decrease the gilby's energy?
Byte 37: Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38: Number of waves in data.
Byte 39: Number of ships in wave.

Planet 4 - Enemy Behaviour Data.

Level	Byte 16	Byte 23	Byte 24	Byte 35	Byte 36	Byte 37	Byte 38	Byte 39
1	$60	$00	$01	$01	$01	$00	$04	$18
2	$00	$00	$23	$00	$08	$00	$04	$18
3	$00	$00	$23	$02	$01	$00	$04	$30
4	$10	$01	$00	$02	$02	$00	$04	$20
5	$30	$00	$00	$02	$02	$00	$04	$20
6	$00	$00	$01	$00	$05	$00	$04	$20
7	$00	$00	$01	$01	$03	$00	$04	$20
8	$00	$00	$23	$00	$10	$00	$04	$30
9	$E0	$00	$00	$02	$01	$00	$04	$08
10	$00	$00	$00	$03	$03	$00	$04	$18
11	$0C	$10	$01	$03	$02	$00	$04	$18
12	$00	$18	$23	$00	$04	$00	$04	$08
13	$00	$00	$00	$00	$20	$00	$04	$C0
14	$00	$00	$01	$03	$01	$00	$04	$60
15	$00	$00	$00	$06	$10	$00	$04	$10
17	$00	$00	$23	$00	$0C	$00	$04	$30
18	$30	$01	$01	$00	$0C	$00	$04	$40
20	$00	$00	$23	$01	$01	$00	$04	$40

Byte 16: Update rate for attack wave
Byte 23: Stickiness factor, does the enemy stick to the player
Byte 24: Does the enemy gravitate quickly toward the player when its hit?
Byte 35: Does destroying this enemy increase the gilby's energy?.
Byte 36: Does colliding with this enemy decrease the gilby's energy?
Byte 37: Is the ship a spinning ring, i.e. does it allow the gilby to warp?
Byte 38: Number of waves in data.
Byte 39: Number of ships in wave.

Planet 5 - Enemy Behaviour Data.

3.0.3 Level Movement Data

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$06	$01	$01	$01
2	$00	$00	$24	$02	$01
3	$00	$FA	$01	$01	$02
4	$00	$07	$00	$01	$02
5	$00	$FC	$23	$02	$02
6	$00	$00	$00	$01	$01
7	$00	$07	$00	$01	$02
8	$00	$05	$00	$01	$02
9	$00	$FC	$23	$01	$03
10	$00	$00	$25	$00	$02
11	$03	$02	$00	$01	$02
12	$00	$FC	$21	$01	$01
13	$00	$00	$24	$02	$02
14	$03	$FA	$00	$01	$01
15	$00	$00	$00	$01	$01
16	$00	$00	$00	$02	$00
17	$00	$80	$80	$01	$01
18	$00	$06	$00	$01	$02
19	$00	$00	$00	$02	$02
20	$00	$04	$24	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Planet 1 - Movement Data.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$00	$00	$01	$02
2	$00	$E9	$00	$01	$02
3	$00	$17	$00	$01	$03
4	$00	$FC	$00	$02	$02
5	$00	$06	$24	$01	$02
6	$00	$07	$24	$01	$01
7	$00	$04	$00	$01	$01
8	$00	$00	$00	$00	$01
9	$00	$04	$24	$01	$02
10	$00	$06	$00	$01	$00
11	$00	$00	$00	$01	$02
12	$00	$03	$00	$01	$00
13	$00	$00	$00	$02	$02
14	$00	$E9	$00	$01	$02
15	$00	$03	$22	$01	$01
16	$00	$FC	$00	$01	$00
17	$00	$17	$00	$01	$03
18	$00	$00	$00	$01	$01
19	$0C	$05	$24	$01	$02
20	$00	$FC	$24	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Planet 2 - Movement Data.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$00	$00	$02	$02
2	$00	$F8	$01	$01	$0C
3	$00	$03	$23	$01	$01
4	$04	$08	$00	$01	$03
5	$00	$00	$00	$00	$00
6	$04	$F9	$23	$01	$07
7	$00	$03	$23	$01	$02
8	$00	$00	$00	$00	$00
9	$00	$00	$00	$01	$02
10	$00	$00	$00	$01	$02
11	$00	$FD	$21	$02	$01
12	$00	$00	$23	$00	$01
13	$00	$00	$00	$00	$00
14	$00	$00	$00	$00	$00
15	$00	$00	$00	$00	$00
16	$00	$00	$00	$00	$00
17	$00	$03	$00	$01	$01
18	$00	$00	$00	$02	$02
19	$00	$05	$00	$01	$02
20	$00	$06	$24	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Planet 3 - Movement Data.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$04	$23	$01	$02
2	$00	$0C	$00	$01	$02
3	$00	$03	$00	$01	$03
4	$00	$00	$00	$00	$01
5	$00	$04	$23	$01	$02
6	$00	$00	$00	$00	$00
7	$00	$00	$00	$00	$00
8	$00	$00	$00	$00	$00
9	$00	$80	$25	$80	$02
10	$00	$0A	$00	$01	$02
11	$00	$00	$00	$00	$01
12	$00	$00	$00	$00	$00
13	$00	$F9	$00	$01	$01
14	$00	$80	$80	$80	$80
15	$00	$00	$00	$02	$03
16	$00	$F8	$00	$01	$04
17	$00	$04	$00	$01	$00
18	$00	$00	$24	$02	$03
19	$01	$01	$00	$01	$01
20	$00	$FA	$24	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Planet 4 - Movement Data.

Level	Byte 7	Byte 19	Byte 20	Byte 21	Byte 22
1	$00	$FC	$00	$01	$02
2	$00	$00	$25	$00	$01
3	$01	$FD	$24	$01	$02
4	$00	$00	$00	$01	$00
5	$05	$07	$03	$01	$01
6	$00	$F4	$00	$01	$02
7	$00	$FE	$00	$01	$01
8	$00	$04	$24	$01	$02
9	$00	$00	$00	$00	$00
10	$00	$00	$20	$00	$01
11	$00	$00	$00	$01	$02
12	$00	$00	$23	$02	$02
13	$00	$00	$00	$00	$00
14	$06	$FC	$00	$01	$02
15	$00	$00	$00	$00	$00
17	$00	$02	$22	$01	$01
18	$00	$00	$00	$02	$02
20	$00	$0C	$24	$01	$02

Byte 7 : Whether a specific attack behaviour is used.
Byte 19: X Pos movement for attack ship.
Byte 20: Y Pos movement pattern for attack ship.
Byte 21: X Pos Frame Rate for Attack ship.
Byte 22: Y Pos Frame Rate for Attack ship.

Planet 5 - Movement Data.

Appendix: Planet Data

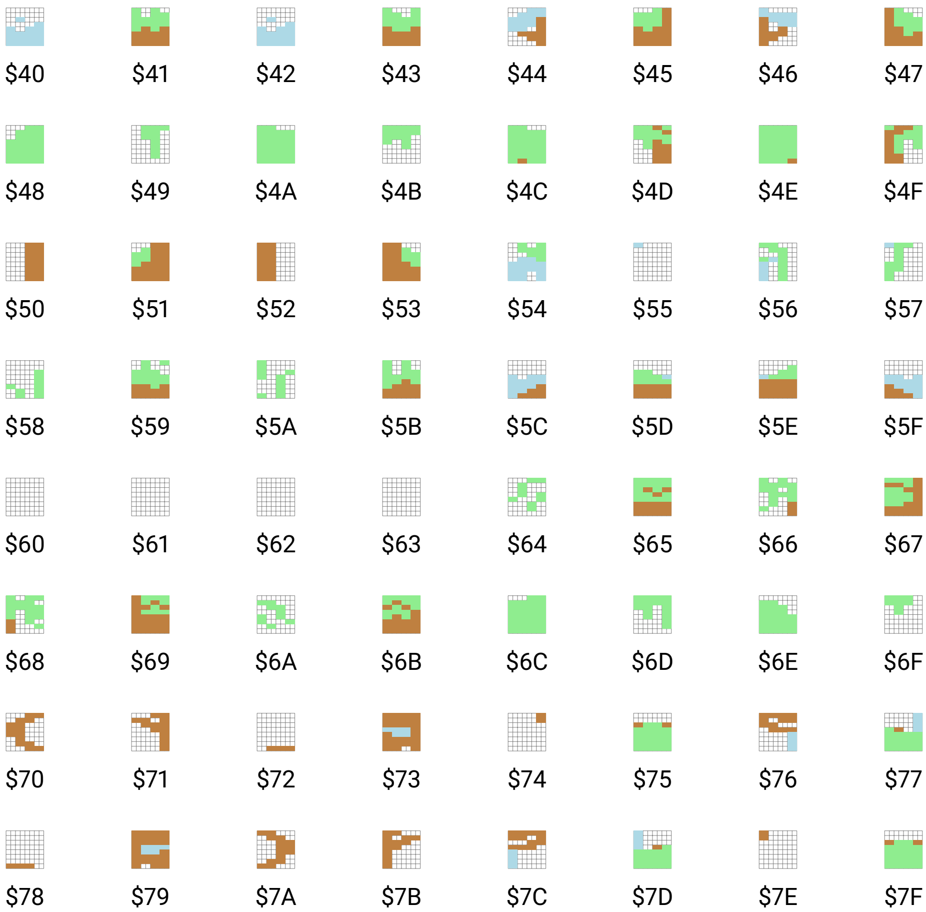


Figure 4.1: Tilesheet: planet1Charset

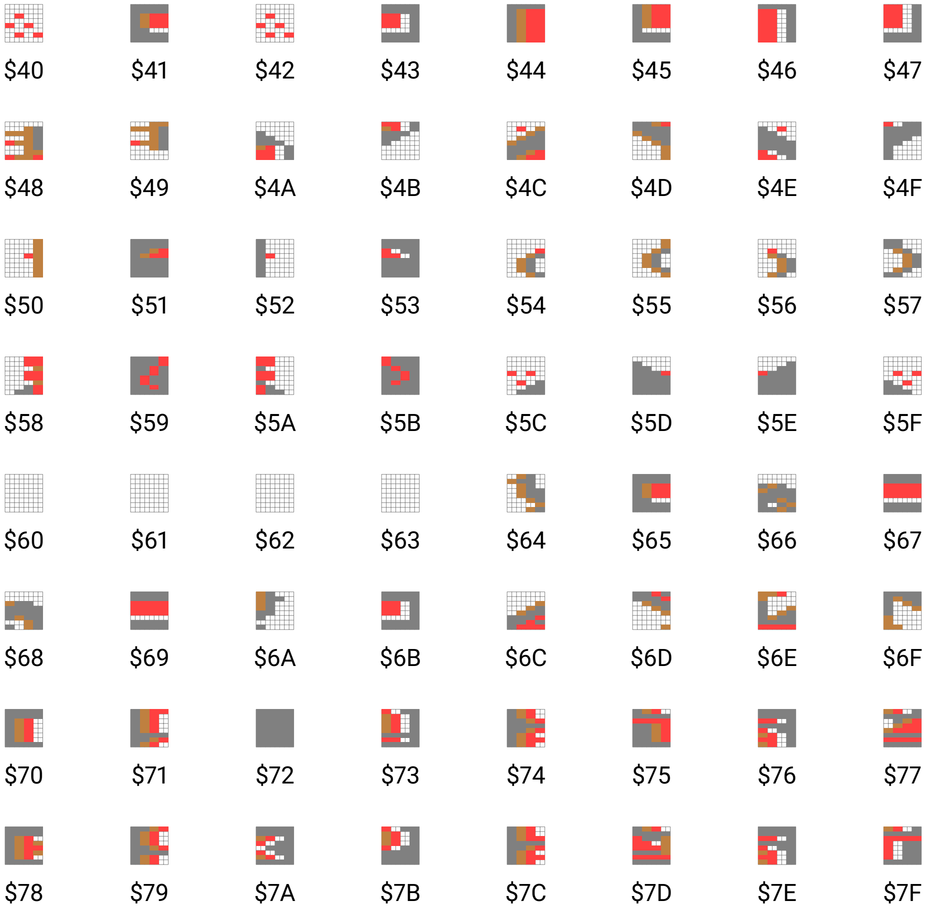


Figure 4.2: Tilesheet: planet2Charset

Figure 4.3: Tilesheet: planet3Charset

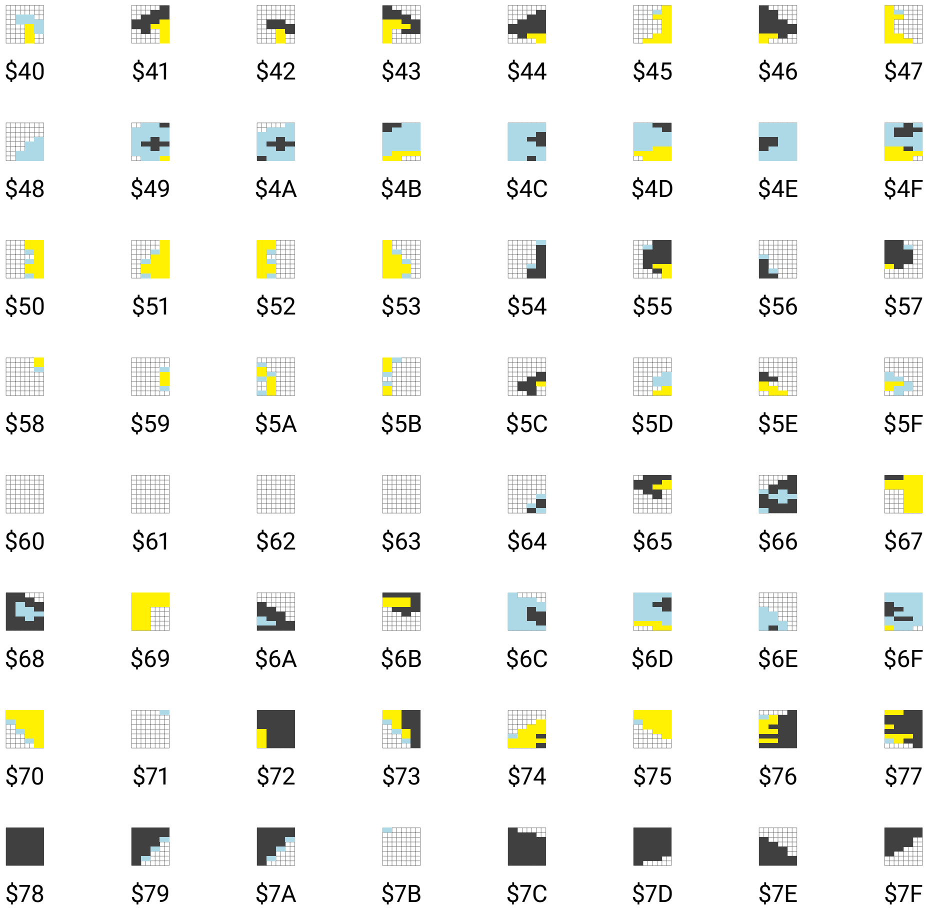


Figure 4.4: Tilesheet: planet4Charset

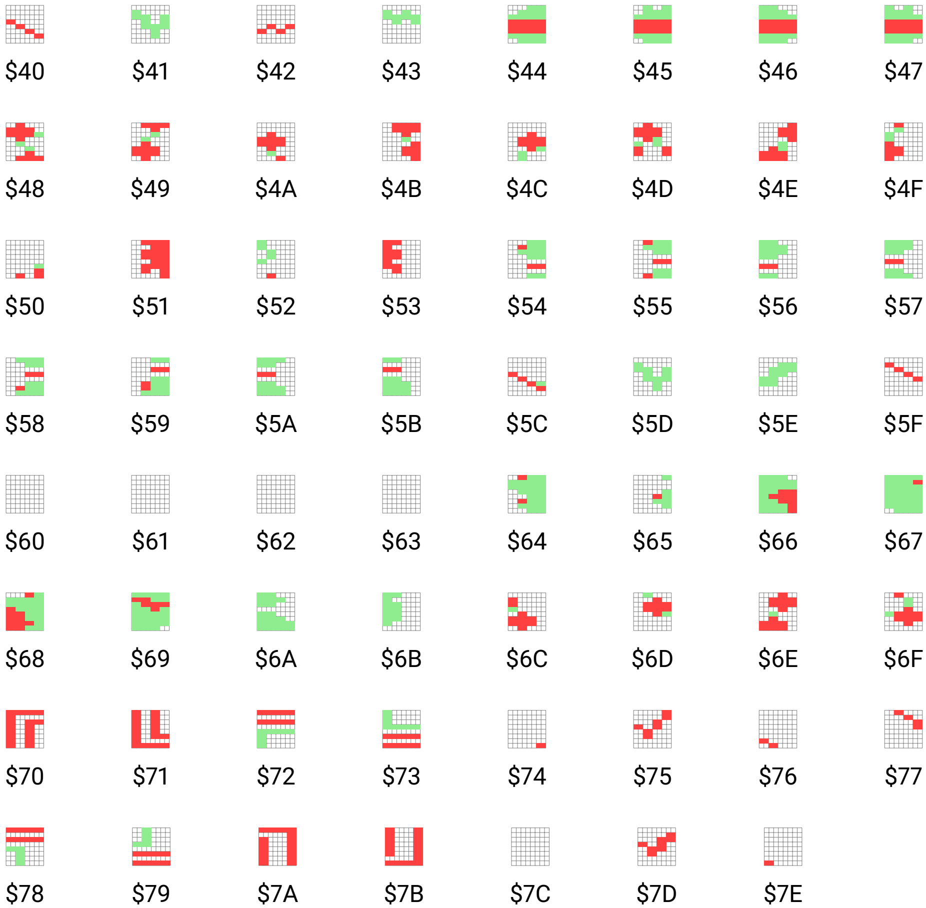
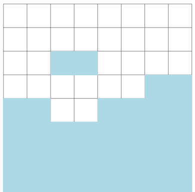
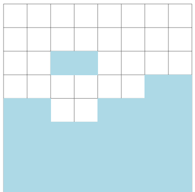


Figure 4.5: Tilesheet: planet5Charset



(a) planet1Charset $40



(b) planet1Charset $42

Figure 4.6: Tilesheet: Planet 1 Sea.

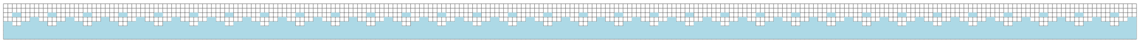
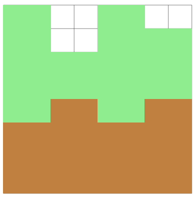
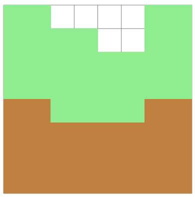


Figure 4.7: planet1Charset Sea



(a) planet1Charset $41



(b) planet1Charset $43

Figure 4.8: Tilesheet: Planet 1 Land.

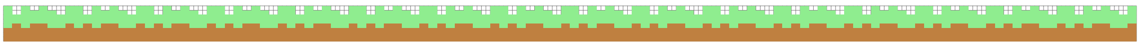
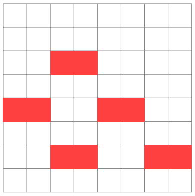
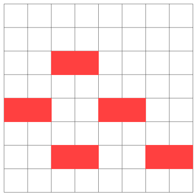


Figure 4.9: planet1Charset Land



(a) planet2Charset $40



(b) planet2Charset $42

Figure 4.10: Tilesheet: Planet 2 Sea.

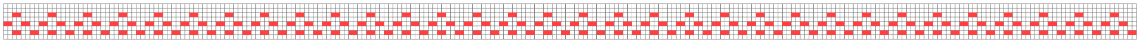
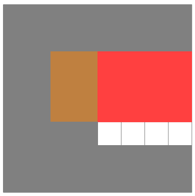
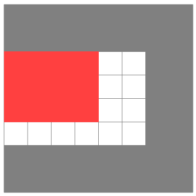


Figure 4.11: planet2Charset Sea



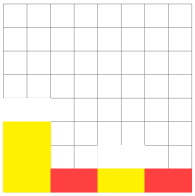
(a) planet2Charset $41



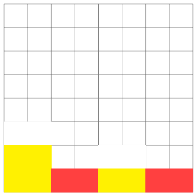
(b) planet2Charset $43

Figure 4.12: Tilesheet: Planet 2 Land.

Figure 4.13: planet2Charset Land



(a) planet3Charset $40



(b) planet3Charset $42

Figure 4.14: Tilesheet: Planet 3 Sea.

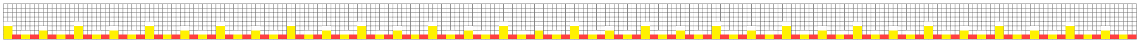


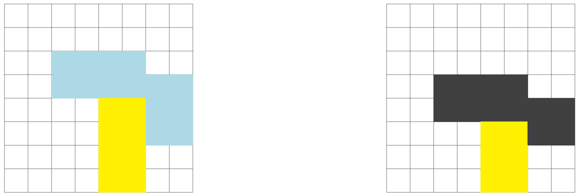
Figure 4.15: planet3Charset Sea

(a) planet3Charset $41

(b) planet3Charset $43

Figure 4.16: Tilesheet: Planet 3 Land.

Figure 4.17: planet3Charset Land



(a) planet4Charset \$40 **(b)** planet4Charset \$42

Figure 4.18: Tilesheet: Planet 4 Sea.

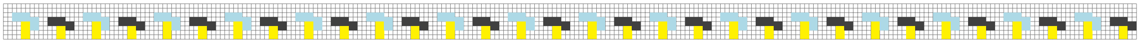


Figure 4.19: planet4Charset Sea

(a) planet4Charset \$41 **(b)** planet4Charset \$43

Figure 4.20: Tilesheet: Planet 4 Land.

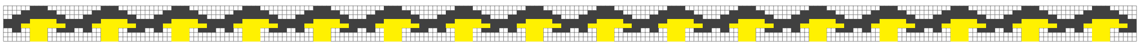
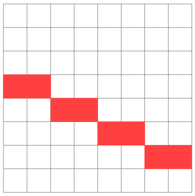
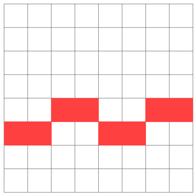


Figure 4.21: planet4Charset Land



(a) planet5Charset $40



(b) planet5Charset $42

Figure 4.22: Tilesheet: Planet 5 Sea.

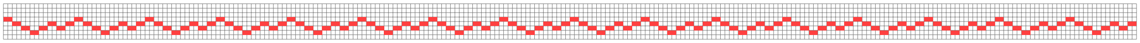
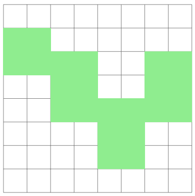
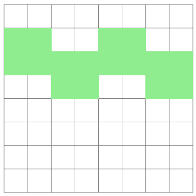


Figure 4.23: planet5Charset Sea



(a) planet5Charset $41



(b) planet5Charset $43

Figure 4.24: Tilesheet: Planet 5 Land.

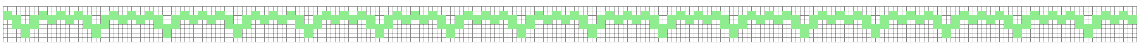


Figure 4.25: planet5Charset Land